Borland® C++ for OS/2®
Version 1.5
Redistributable files
You can redistribute the following files in accordance with the No Nonsense License Statement:

- BIDS402.DLL
- BIDS402D2.DLL
- TCLASS2.DLL
- C215MT.DLL
- C215.DLL
- BPMCC.DLL
- LOCALE.BLL

Borland may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1987, 1994 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

Borland International, Inc.
100 Borland Way, Scotts Valley, CA 95066-3249

PRINTED IN THE UNITED STATES OF AMERICA
# Contents

## Introduction
- What's in Borland C++ ........................................ 1
- Hardware and software requirements ................... 2
- The Borland C++ implementation ....................... 3
- The Borland C++ package ................................ 3
  - The User's Guide ....................................... 3
  - The Tools and Utilities Guide ......................... 4
  - The Programmer's Guide ................................ 5
  - The Library Reference .................................. 6
- Typefaces and icons used in these books ............... 6
- Tools in your package .................................... 7
- Contacting Borland ........................................ 8
  - Borland Assist plans .................................. 8

### Chapter 1 Installing Borland C++
- Using INSTALL ............................................. 11
  - Running the IDE ....................................... 13
- Opening the README file ................................ 13
- The HELPME!.DOC file .................................... 13
- Customizing the IDE ...................................... 14
- Sample programs ......................................... 14

### Chapter 2 IDE basics
- Starting the IDE ........................................... 15
  - Startup options ........................................ 17
  - The /b option ........................................... 17
  - The /m option ........................................... 17
- Exiting the IDE .......................................... 17
- IDE components ........................................... 18
  - The menu bar and menus ................................. 18
  - Mouse shortcuts ........................................ 19
  - Using the SpeedBar ..................................... 19
  - Keyboard shortcuts .................................... 21
  - Borland C++ windows .................................... 23
  - Window management .................................... 25
  - The status line ........................................ 26
  - Dialog boxes ........................................... 26
  - Action buttons .......................................... 26
  - Radio buttons and check boxes ....................... 27
  - Input and list boxes .................................. 27
- Configuration and project files ......................... 28
  - The configuration file ................................ 28
  - Project files ........................................... 28
    - The project directory ................................. 29
    - Desktop files ....................................... 29

## Default files ............................................. 29
- Changing project files .................................. 30
- Syntax highlighting ...................................... 30
- Configuring element colors ................................ 30

## Some basic tasks ........................................ 31
- Compiling and linking programs .......................... 31
  - Making an application ................................ 31
  - Building an application ................................ 31
  - Compiling a file ....................................... 31
  - Linking a file .......................................... 32
- Debugging an application .................................. 32
  - Preparing your application ............................ 32
  - Debugging environment ................................ 33
  - Viewing data objects .................................. 34
  - Controlling program execution ........................ 34

### Chapter 3 Menus and options reference
- File menu .................................................. 37
  - New ....................................................... 37
  - Open ...................................................... 37
  - Using the File list box ................................ 38
  - Save ....................................................... 38
  - Save As .................................................. 38
  - Save All ................................................ 38
  - Print ..................................................... 39
  - Exit ....................................................... 39
  - Closed File Listing .................................... 39
- Edit menu .................................................. 39
  - Undo ....................................................... 41
  - Redo ....................................................... 41
  - Cut ....................................................... 41
  - Copy ....................................................... 41
  - Paste ..................................................... 41
  - Clear ..................................................... 41
- Search menu ............................................... 41
  - Find ....................................................... 41
  - Replace .................................................. 43
  - Search Again ............................................ 44
  - Go to Line Number ....................................... 44
- Run menu .................................................. 44
  - Run ....................................................... 44
  - Using the same source code ............................ 44
  - Using modified source code ........................... 44
  - Step Over ............................................... 45
Appendix A The optimizer
What is optimization? 129
When should you use the optimizer? 129
Optimization options 129
A closer look at the Borland C++ optimizer 131
Global register allocation 131
Global optimizations 131
Common subexpression elimination 131
Loop invariant code motion 132
Copy propagation 132
Induction variable analysis and strength reduction 133
Linear function test replacement 133
Loop compaction 134
Dead storage elimination 135
Pointer aliasing 135
Code size versus speed optimizations 136
Intrinsic function inlining 136
Register parameter passing 138
Parameter rules 138
Floating-point registers 138
Function naming 139

Appendix B Editor reference
Block commands 144
Other editing commands 145

Appendix C Precompiled headers
How they work 147
Drawbacks 148
Using precompiled headers 148
Setting file names 148
Establishing identity 148
Optimizing precompiled headers 149

Appendix D Using the Browser
Browsing through your code 151
Browsing through objects 153
Filters 154
Viewing declarations of listed symbols 155
Browsing through global symbols 155
Browsing symbols in your code 155

Index 157
<table>
<thead>
<tr>
<th>Tables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 General hot keys</td>
<td>22</td>
</tr>
<tr>
<td>2.2 Menu hot keys</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Editing hot keys</td>
<td>23</td>
</tr>
<tr>
<td>2.4 Online Help hot keys</td>
<td>23</td>
</tr>
<tr>
<td>2.5 Debugging/Running hot keys</td>
<td>23</td>
</tr>
<tr>
<td>2.6 Manipulating windows</td>
<td>25</td>
</tr>
<tr>
<td>3.1 Search-string wildcards</td>
<td>42</td>
</tr>
<tr>
<td>6.1 Command-line options summary</td>
<td>106</td>
</tr>
<tr>
<td>A.1 Optimization options summary</td>
<td>130</td>
</tr>
<tr>
<td>A.2 Parameter types and possible registers used</td>
<td>138</td>
</tr>
<tr>
<td>B.1 Editing commands</td>
<td>141</td>
</tr>
<tr>
<td>B.2 Block commands in depth</td>
<td>144</td>
</tr>
<tr>
<td>B.3 Borland-style block commands</td>
<td>145</td>
</tr>
<tr>
<td>B.4 Other editor commands in depth</td>
<td>145</td>
</tr>
<tr>
<td>D.1 Letter symbols in the Browser</td>
<td>154</td>
</tr>
</tbody>
</table>
Figures

D.1 Buttons on the Browser SpeedBar .......... 152  D.3 Viewing the details of an object .......... 154
D.2 Viewing the object hierarchy of an
application .................................... 153
Introduction

Borland C++ is a professional optimizing compiler for C++ and C developers. It's powerful, fast, and efficient. With Borland C++, you can create practically any OS/2 or Presentation Manager application.

Because C++ is an object-oriented programming (OOP) language, it gives you the advantages of advanced design methodology and labor-saving features. It's the next step in the natural evolution of C. And because it's portable, you can easily transfer application programs written in C++ from one system to another. You can use C++ for almost any programming task on any platform.

What's in Borland C++

Borland C++ includes the latest features programmers have asked for:

- **C and C++**: Borland C++ offers you the full power of C and C++ programming, with a complete implementation of the AT&T v. 3.0 specification as well as a 100% ANSI C compiler. Borland C++ for OS/2 also provides a number of useful C++ class libraries, plus the a complete implementation of templates and exception handling, which allow efficient collection classes to be built using parameterized types.

- **Global optimization**: a full suite of state-of-the-art optimizations gives you complete control over code generation, so you can program in the style you find most convenient, yet still produce small, fast, highly efficient code.

- **Faster compilation speed**: Precompiled headers significantly shorten recompilation time. Optimizations are also performed at high speed, so you don’t have to wait for high quality code.

- **Programmer’s Platform**: Borland C++ for OS/2 comes with an improved version of the Programmer's Platform, Borland's open-architecture integrated development environment (IDE) that gives you access to a full range of programming tools and utilities, including
  
  - Turbo Editor Macro Language (TEML) and the Turbo Editor Macro Compiler (TEMC), which provide the ability to create and use a customized editor interface.
• Multiple overlapping windows with full mouse support.
• Integrated resource compiling and linking.
• Fully integrated debugger with support for multi-thread debugging.
• Support for inline assembly code.
• Complete undo and redo capability with a large buffer.
• Built-in Browser that lets you visually explore your class hierarchies, functions and variables, locate inherited function and data members, and instantly browse the source code of any element you select.
• Visual SpeedBar for instant point-and-click access to frequently used menu selections.

■ Help: Online context-sensitive hypertext help, with copy-and-paste program examples for almost every function. You can reach the help functions from anywhere in the IDE by simply pressing F1.

■ Streams: Full support for C++ iostreams, plus special Borland extensions to the streams library that let you position text, set screen attributes, and perform other manipulations to streams within the OS/2 environment.

■ Container classes: Advanced container class libraries giving you sets, bags, lists, arrays, B-trees, and other reusable data structures. The containers are implemented as templates.

■ OS/2 API: The complete OS/2 API documentation in online Help.

Other features:

■ Over 200 extended library functions for maximum flexibility and compatibility.
■ Complex and binary-coded decimal (BCD) math.
■ Response files for the command-line compiler.
■ NMAKE compatibility for easy transition from Microsoft C or C++.

**Hardware and software requirements**

Borland C++ runs on the IBM PS/2- and PC-compatible family of computers running the OS/2 operating system. Borland C++ requires OS/2 2.1 or higher, 28M of hard disk space, a floppy drive, and at least 6M of memory; it runs on any OS/2-compatible monitor.

Borland C++ includes floating-point routines that let your programs make use of an 80x87 math coprocessor chip. It emulates the chip if it is not available. Though it is not required to run Borland C++, the 80x87 chip can
significantly enhance the performance of your programs that use floating-point math operations.

The Borland C++ implementation

Borland C++ is a full implementation of the AT&T C++ version 3.0 with exception handling. It also supports the American National Standards Institute (ANSI) C standard. In addition, Borland C++ includes certain extensions for mixed-language programming that let you exploit your PC’s capabilities. See Chapters 1–5 in the Programmer’s Guide for a complete formal description of Borland C++.

The Borland C++ package

Your Borland C++ package consists of a set of disks and eight manuals.

The disks contain all the programs, files, and libraries you need to create, compile, link, and run your Borland C++ programs; they also contain sample programs, many standalone utilities, a contextual help file, an integrated debugger, and C and C++ documentation in online text files.

These are the eight manuals:

- Borland C++ User’s Guide
- Borland C++ Tools and Utilities Guide
- Borland C++ Library Reference
- Borland C++ Programmer’s Guide
- Resource Workshop User’s Guide
- Turbo Debugger User’s Guide
- Turbo Assembler User’s Guide
- Turbo Assembler Quick Reference

The User’s Guide introduces you to Borland C++ and shows you how to create and run both C and C++ programs. It consists of information you’ll need to get up and running quickly, and provides reference chapters on the features of Borland C++: the Programmer’s Platform—including the editor and Project Manager—and the command-line compiler. These are the chapters in this manual:

Introduction introduces you to Borland C++ and tells you where to look for more information about each feature and option.
**Chapter 1: Installing Borland C++** tells you how to install Borland C++ on your system; it also tells you how to configure your installation, defaults, and many other aspects of Borland C++.

**Chapter 2: IDE basics** introduces the features of the Programmer's Platform, giving information and examples of how to use the IDE to full advantage. It includes information on how to start up and exit from the IDE, descriptions of the IDE's local menus (which provide a large part of the IDE's functionality), and describes basic programming and debugging techniques within the IDE.

**Chapter 3: Menus and options reference** provides a complete reference to the menus and options in the Programmer's Platform.

**Chapter 4: Settings notebook** explains the use of the Settings notebook for setting the various compilation, linking, and environment settings available in the IDE.

**Chapter 5: Managing multi-file projects** introduces you to Borland C++'s built-in project manager and shows you how to build and update large projects from within the IDE.

**Chapter 6: Command-line compiler** explains the use of the command-line compiler. It also explains how to use compiler configuration files.

**Appendix A: The optimizer** introduces the concepts of compiler optimization, and describes the specific optimization strategies and techniques available in Borland C++.

**Appendix B: Editor reference** provides a convenient command reference to using the editor with the CUA command interface.

**Appendix C: Precompiled headers** tells you how to use Borland C++'s precompiled headers feature to save substantial time when recompiling large projects.

**Appendix D: Using the Browser** tells you how to use the IDE Browser to explore objects hierarchies, functions, and variables in your program.

The *Tools and Utilities Guide* introduces you to the many programming tools and utility programs provided with Borland C++. It contains information you’ll need to make full use of the Borland C++ programming environment, including the Make utility, the Turbo Librarian and Linker, and special utilities for PM programming.

**Chapter 1: TLINK: The Turbo linker** is a complete reference to the features and functions of the Turbo Linker (TLINK).
Chapter 2: Make: The program manager introduces the Borland C++ MAKE utility, describes its features and syntax, and presents some examples of usage.

Chapter 3: TLIB: The Turbo librarian tells you how to use the Borland C++ Turbo Librarian to combine object files into integrated library (.LIB) files.

Chapter 4: Import library tools tells you how to use the IMPDEF and IMPLIB utilities to define and specify import libraries.

Chapter 5: Resource tools tells you how to use the Resource Compiler to compile .RC scripts into .RES resource files for your PM programs.

Appendix A: Error messages lists and explains run-time, compile-time, linker, and librarian errors and warnings, with suggested solutions.

The Programmer's Guide provides useful material for the experienced C user: a complete language reference for C and C++, writing PM applications, C++ streams, Borland C++ class libraries, OS/2 memory management, and floating-point issues.

Chapters 1–5: Lexical elements, Language structure, C++ specifics, Exception handling, and The preprocessor describe the Borland C++ language.

Chapter 6: Using C++ streams tells you how to use the C++ iostreams library, as well as special Borland C++ extensions for PM.

Chapter 7: Using Borland class libraries tells you how to use the Borland C++ container class library in your programs.

Chapter 8: Dynamic-link libraries discusses how to build and use dynamic-link libraries under OS/2.

Chapter 9: Building OS/2 applications introduces you to the concepts and techniques of writing applications for PM using Borland C++.

Chapter 10: Mathematical operations covers floating-point, BCD, and complex math.

Chapter 11: OS/2 memory management describes the OS/2 memory-management scheme and system calls.

Chapter 12: Inline assembly tells how to write inline assembly language functions within your Borland C++ program.

Appendix A: ANSI implementation-specific standards describes those aspects of the ANSI C standard that have been left loosely defined or undefined by ANSI, and how Borland has chosen to implement them.
The Library Reference contains a detailed list and explanation of Borland C++'s extensive library functions and global variables.

Chapter 1: The main function describes the main function.

Chapter 2: Run-time functions is an alphabetically arranged reference to all Borland C++ library functions.

Chapter 3: Global variables defines and discusses Borland C++'s global variables.

Chapter 4: The C++ iostreams provides a reference to the C++ iostreams library, including the Borland extensions to the library.

Chapter 5: Persistent stream classes and macros describes the persistent streams classes and macros.

Chapter 6: The C++ container classes provides a reference to the Borland implementation of the container class library.

Chapter 7: The C++ mathematical classes describes the Borland implementation of the C++ math class libraries.

Chapter 8: Class diagnostic macros describes the classes and macros that support object diagnostics.

Chapter 9: Run-time support describes functions and classes that let you control the way your program executes at run time in case the program runs out of memory or encounters some exception.

Chapter 10: C++ utility classes describes the C++ date, string, and time classes.

Appendix A: Run-time library cross-reference provides a complete indexed locator reference to all Borland C++ library functions.

Typefaces and icons used in these books

All typefaces and icons used in this manual were produced by Borland's Sprint: The Professional Word Processor, on a PostScript laser printer.

Monospaced type This typeface represents text as it appears onscreen or in a program. It is also used for anything you must type literally (such as `BC` to start up the Borland C++ IDE).

ALL CAPS The names of constants and files (except for header files) are spelled with all capital letters.
[] Square brackets [ ] in text or OS/2 command lines enclose optional items that depend on your system. *Text of this sort should not be typed verbatim.*

<> Angle brackets in the function reference section enclose the names of include files.

**Boldface** This typeface is used in text for Borland C++ reserved words (such as `char`, `switch`, `void`, and `__cdecl`), for format specifiers and escape sequences (%d, \t), and for command-line options (/b).

*Italic* Borland C++ function names (such as `printf`), class, and structure names are shown in italics when they appear in text (but not in program examples). *Italics* also indicate variable names (identifiers) that appear in text. They can represent terms that you can use as is, or that you can think up new names for (your choice, usually). Italic type is also used to emphasize certain words, such as new terms.

**Keycaps** This typeface indicates a key on your keyboard. For example, “Press Esc to exit a menu.”

**Initial Caps** Menu choices and items in dialog boxes are indicated by capitalizing the first letter of each word.

This icon indicates keyboard actions.

This icon indicates mouse actions.

This icon indicates language items that are specific to C++. It is used primarily in the *Programmer’s Guide*.

**Tools in your package**

This product contains many tools to help you:

- The manuals provide information on every aspect of the program. Use them as your main information source.

- While using the IDE, you can press F1 for general help, Ctrl+F1 for help about the currently selected item, or Shift+F1 for an index of topics in the Help system.

- If you are using the command-line compiler, use the OS/2 utility VIEW for online help. For information on VIEW, see your OS/2 documentation.

- Many common questions are answered in the DOC files listed in the README file located in the installation directory of your Borland compiler.
Contacting Borland

The Borland Assist program offers a range of services to fit the different needs of individuals, consultants, large corporations, and developers. To receive help with your questions about our products, send in the registration card. North American customers can register by phone 24 hours a day by calling 1-800-845-0147.

Borland Assist is made up of three levels of support:

- **Standard Assist** gives all registered users assistance with installation and configuration, and offers automated and online services to answer other product questions (see the following table).
- **Enhanced Assist** plans are designed for individuals who need unlimited support on a toll-free number or priority hotline access.
- **Premium Assist** plans are designed to support large corporations and software developers.

Available at no charge, **Standard Assist** offers all registered users the following services:

<table>
<thead>
<tr>
<th>Service</th>
<th>How to contact</th>
<th>Cost</th>
<th>Available</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation</td>
<td>408-461-9133</td>
<td>The cost of the phone call</td>
<td>6:00am – 5:00pm PST Monday – Friday</td>
<td>Provides assistance on product installation and configuration.</td>
</tr>
<tr>
<td>hotline</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automated</td>
<td>Voice: 1-800-524-8420 Modem: 408-431-5250</td>
<td>Free</td>
<td>24 hours daily</td>
<td>Provides answers to common questions Requires a Touch-Tone phone or modem.</td>
</tr>
<tr>
<td>support</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TechFax</td>
<td>1-800-822-4269 (voice)</td>
<td>Free</td>
<td>24 hours daily</td>
<td>Sends technical information to your fax machine (up to 3 documents per call). Requires a Touch-Tone phone. Document #1 is the catalog of available catalogs.</td>
</tr>
</tbody>
</table>

**Online services**

<table>
<thead>
<tr>
<th>Service</th>
<th>How to contact</th>
<th>Cost</th>
<th>Available</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Borland</td>
<td>408-431-5096</td>
<td>The cost of the phone call</td>
<td>24 hours daily</td>
<td>Sends sample files, applications, and technical information via your modem. Requires a modem (up to 9600 baud).</td>
</tr>
<tr>
<td>Download BBS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CompuServe</td>
<td>Type GO BORLAND. Address messages to Sysop or All.</td>
<td>Your online charges</td>
<td>24 hours daily; 1-working-day response time</td>
<td>Sends answers to technical questions via your modem. Messages are public.</td>
</tr>
<tr>
<td>BIX</td>
<td>Type JOIN BORLAND. Address messages to Sysop or All.</td>
<td>Your online charges</td>
<td>24 hours daily; 1-working-day response time</td>
<td>Sends answers to technical questions via your modem. Messages are public.</td>
</tr>
</tbody>
</table>
For additional details on these and other Borland services, see the *Borland Assist Support and Services Guide* included with your product.
Your Borland C++ package includes two different versions of Borland C++: the IDE (Programmer's Platform) and the OS/2 command-line version.

If you don't already know how to use OS/2 commands, refer to your OS/2 reference manual before setting up Borland C++ on your system.

Borland C++ comes with an automatic installation program called INSTALL. Because we used file-compression techniques, you must use this program; you can't just copy the Borland C++ files onto your hard disk. INSTALL automatically copies and decompresses the Borland C++ files. FILELIST.DOC on the installation disk includes a list of the distribution files, with a brief description of what each one contains.

We assume you're already familiar with OS/2 commands. For example, you'll need the DISKCOPY command to make backup copies of your distribution disks. Make a complete working copy of your distribution disks when you receive them, then store the original disks away in a safe place.

This chapter contains the following information:

- How to use INSTALL.
- How to access the README file.
- How to access the HELPME! file.
- Pointers to more information on Borland's sample programs.
- Information about customizing Borland C++ (setting or changing defaults, colors, and so on).

### Using INSTALL

INSTALL detects what hardware you are using and configures Borland C++ appropriately. It also creates directories as needed and transfers files from your distribution disks (the disks you bought) to your hard disk.

To install Borland C++, follow these steps:

1. Insert the installation disk (disk 1) into drive A:
2. Click the icon for drive A:
3. Click the Install icon.
4. The Borland C++ For OS/2 Installation dialog box opens up. It has eight controls:

- Installation Options lets you specify which parts of the Borland C++ package you want to install, whether the installation program should create a PM program group for the compiler, and whether the installation program should modify your CONFIG.SYS file to support the Borland C++ compiler.
- Directory Options lets you specify the directories where you want each part of the compiler installed. By default, these are subdirectories below the directory specified in the Base Directory input box.
- Base Directory lets you specify the name of the directory in which you want the compiler installed.
- Install From lets you specify where the Borland C++ installation files are located.
- Install starts the installation procedure based on the options you define through the other controls in this dialog box.
- Exit exits the installation procedure without installing the compiler.
- Reset resets all options to their default state (the state they were in when you first ran the installation program).
- Help gives you help in installing the Borland C++ package.

Use these controls to configure your Borland C++ installation to your satisfaction, then click Install to begin installing the compiler.

5. If you did not tell INSTALL to modify your CONFIG.SYS file in the Installation Options dialog box, you must make the following changes to your CONFIG.SYS file for the compiler to function correctly:

- Modify the PATH line in your CONFIG.SYS file to contain the directory where your compiler is installed:

  ```
  PATH=C:\OS2;C:\OS2\SYSTEM; ... ;C:\BORLANDC\BIN
  ```

  where BORLANDC is the name of the directory where you installed Borland C++.

- Modify the LIBPATH line in your CONFIG.SYS file to contain the directory where your compiler is installed:

  ```
  LIBPATH=C:\OS2;C:\OS2\SYSTEM; ... ;C:\BORLANDC\BIN
  ```

  where BORLANDC is the name of the directory where you installed Borland C++.

LIBPATH points to the directory containing all the DLLs for the compiler, linker, and debugger. If PATH is set correctly, but LIBPATH is not, the Borland C++ tools will not work.
6. Reboot your machine so the changes in the PATH and LIBPATH variables take effect.

**Important!** When the installation process is complete, INSTALL opens the README file for you to read. The README file contains important, last-minute information about Borland C++.

After you exit the README file, INSTALL creates a Borland C++ program group and installs it on your desktop if you chose the Create Borland C++ Program Group option in the Borland C++ For OS/2 Installation dialog box. The program group contains icons for the following Borland C++ programs and utilities:

- Borland C++
- Turbo Debugger
- Resource Workshop
- Import Librarian

**Note** If you reinstall your compiler in the future, OS/2 replaces the existing icons with new ones.

---

**Running the IDE**

If you’re anxious to get up and running once you’ve installed Borland C++, start by opening the Borland C++ folder and clicking on the Borland C++ icon. This starts up the Borland C++ Programmer’s Platform, or IDE. For help in the IDE, press F1.

---

**Opening the README file**

Borland C++ automatically places you in the README file when you run the INSTALL program. To access the README file at a later time, open the README file using any regular OS/2 text editor. The file is located in the root directory of your compiler installation.

---

**The HELPME!.DOC file**

Your installation disk contains a file called HELPME!.DOC, which contains answers to problems that users commonly run into. Consult it if you find yourself having difficulties. To access the HELPME!.DOC file, open the file using any regular OS/2 text editor. The file is located in the DOC directory of your compiler installation.
Customizing the IDE

Borland C++ lets you completely customize your tools from within the IDE itself, using the various settings that appear in the Settings notebook. These settings let you specify editing modes, default directories, compiler settings, linker options, and much more.

For information on accessing menus and options in the Borland C++ IDE, see Chapter 2, “IDE basics.” For specific information about each menu item, see Chapter 3, “Menus and options reference.” For information about the Settings notebook, see Chapter 4, “Settings notebook.”

Sample programs

Your Borland C++ package includes the source code for a large number of C and C++ sample programs for OS/2. These programs are located in the EXAMPLES directory (and subdirectories) created by INSTALL. Before you compile any of these sample programs, you should read the printed or online documentation for them.

Many of these examples are ported from the IBM OS/2 2.0 Toolkit examples. Comparing the examples provided with Borland C++ with those from the IBM Toolkit can give you some idea of how easy it is to port your programs designed to be compiled with the IBM C Set/2 tools.
IDE basics

Borland's Programmer's Platform, also known as the integrated development environment, or IDE, has everything you need to write, edit, compile, link, and debug OS/2 or Presentation Manager programs. It provides:

- Multiple, movable, resizable windows.
- Language syntax highlighting with customizable colors.
- Cut, paste, and copy commands that use the Clipboard.
- Full editor undo and redo.
- Online Help.
- Examples to copy and paste from the online Help system.
- Inline assembler.
- Quick spawning of other programs.
- Editor macro language.
- Background compilation that lets you perform other tasks during program builds.
- Full built-in debugging capability, including multi-thread support.

This chapter explains how to start up and exit the Borland C++ IDE, discusses its components, describes the options available for both the IDE and the command-line compiler, and explains how configuration and project files work.

Starting the IDE

To start the IDE, you can either double-click the Borland C++ icon or type BC on the OS/2 command line. You can also specify an optional parameter by either typing the parameter on the command line or by configuring the Borland C++ icon with the OS/2 Settings notebook. To open the OS/2 Settings notebook, right-click the BC icon. Click the arrow next to the Open menu choice. When the submenu opens, click Settings. When the Settings notebook opens, click in the Parameters box and type the desired parameter.
Valid parameters for the Borland C++ for OS/2 IDE are /b and /m, along with one or more file or project names. You can also specify one or more file or project names without a /b or /m parameter.

If you specify a name without an extension, Borland C++ assumes it is a source file with the default extension .CPP and opens the file, even if it does not exist. If you specify the name of a project file, Borland C++ opens that project.

If you do not specify a project name, and if there is a single project file with a .PRJ extension in the current directory, Borland C++ automatically opens the project. If there is more than one file with a .PRJ extension in the current directory, Borland C++ doesn’t open any of the projects.

You can set up multiple project icons so that you can load, build, or make various projects by simply clicking on an icon.

To create a new icon for a project, follow these steps:

1. Open the OS/2 Templates folder by double-clicking on it.
2. Right-click the Program icon. Hold the button down.
3. Drag the icon onto the desktop or into a folder and let go of the mouse button.
4. The Settings notebook for the icon automatically opens. Click in the box labeled Path and file name. Type in the path and name of BC.EXE, including the extension.
5. Click in the box labeled Parameters. Type /b or /m if you want to do a build or a make, respectively. Type in the path to the file or files you want to use. If a file is a project file, you must specify the .PRJ extension. If you do not specify /b or /m, BC loads the files or projects you specify.
6. Click the tab labeled General. In the box labeled Title, type in a name for your project. This name helps you distinguish between separate projects, but has no other significance.
7. Double-click the system menu button in the upper left corner to close the Settings notebook. The new icon appears on the desktop with the name you gave it.

You can also work in other OS/2 applications while the IDE is running, even while it is performing a task such as a build or a compile. You can do this in one of several ways:
Startup options

The valid startup options for Borland C++'s IDE are /b and /m, which use this syntax:

```
BC [option] [sourcename | projectname [sourcename]]
```

where `option` can be either /b or /m, `sourcename` is any ASCII file (default extension assumed), and `projectname` is your project file (it must have the .PRJ extension).

The /b (build) option causes Borland C++ to open the IDE, recompile and link all the files in your project, print out all compiler messages, and then close the IDE.

To specify a project file, enter the `BC` command followed by /b and the project file name. For example,

```
BC /b myprog
```

If there is no MYPROG.PRJ file, the following command loads the file MYPROG.CPP in the editor and then compiles and links it:

```
BC /b myprog
```

The /m option lets you do a make rather than a build. That is, only outdated source files in your project are recompiled and linked. Follow the instructions for the /b option, but use /m instead.

Exiting the IDE

There are three ways to leave the IDE completely:

- Choose File | Exit.
- Double-click the system menu button, located in the upper-left corner of the IDE window.
- Press Alt+F4.

You'll be prompted to save your files before exiting, if you haven't already done so.
IDE components

There are three visible components to the IDE desktop: the menu bar at the top, the window area in the middle, and the status line at the bottom. Many menu items also offer dialog boxes.

The menu bar and menus

The menu bar is your primary access to all the menu commands. The menu bar is always visible.

You can choose commands with a mouse in one of two ways:

- Click the desired menu title to display the menu and click the desired command.
- Drag straight from the menu title down to the menu command. Release the mouse button on the command you want (if you change your mind, just drag off the menu; no command is chosen).

If a menu command is followed by an ellipsis (…), choosing the command displays a dialog box. If the command is not followed by an ellipsis, an action occurs as soon as you choose the command.

You can also use the mouse to access local menus throughout the IDE. Click the right mouse button anywhere on the IDE desktop and select a command from the menu that appears. See page 19 for more information on local menus.

Here is how you choose menu commands using the keyboard:

1. Press Alt or F10. This makes the menu bar active; the next thing you type relates to the items on the menu bar.

2. Use the arrow keys to select the menu you want to display, then press Enter.

   As a shortcut for this step, you can just press the underlined letter of the menu title. For example, when the menu bar is active, press E to move to and display the Edit menu. At any time, press Alt and the underlined letter (such as Alt+E) to display the menu you want.

3. Use the arrow keys to select a command from the menu you’ve opened, or press the underlined letter in the command name. Then press Enter.

   At this point, Borland C++ either carries out the command or displays a dialog box.
The IDE makes some menu commands unavailable when it would make no sense to choose them. However, you can always get online Help about currently unavailable commands.

You can also access local menus using the keyboard. Press *Shift+F10*, use the arrow keys to select a command from the menu that appears, and press *Enter*. See the following section for more information on local menus.

**Mouse shortcuts**

Borland C++ offers a number of quick ways to choose menu commands. The click-drag method of selecting a menu item is an example. You can also use the right mouse button as a shortcut for performing a number of tasks. Just right-click anywhere on the IDE desktop or press *Shift+F10*. A local menu appears. Choose a command from the menu by clicking it with the mouse or by using the arrow keys to select a command and pressing *Enter*. The command then executes.

The particular local menu that appears depends on which window is active. There are different local menus for each of four different window types: Edit, Transcript, Project, and desktop. To familiarize yourself with the local menus and the capabilities they provide, try opening a local menu in each new kind of window you encounter.

Menu choices from local menus are referenced by the same notation that is used for menu choices, except with the window type specified before Local. For example, Edit Local | Toggle Breakpoint means you should open a local menu in an edit window and choose the Toggle Breakpoint command.

**Using the SpeedBar**

Borland C++ has a SpeedBar you can use as a quick way to choose menu commands and other actions with the mouse. The first time you start the IDE, the SpeedBar is a horizontal grouping of buttons just under the menu bar. You can use it as it is, change it to be a vertical bar that appears on the left side of the Borland C++ desktop window, or change it to be a floating palette you can move anywhere on the IDE desktop. You can also turn it off. To configure the SpeedBar, turn to the Environment | Desktop subsection of the Settings notebook and select the setting you want.

The buttons on the SpeedBar represent menu commands. They are shortcuts for your mouse, just as certain key combinations are shortcuts when you use your keyboard. To choose a command, click a button with your mouse. If you click the File | Open button, for example, Borland C++ responds just as if you chose the Open command on the File menu.

The SpeedBar is context sensitive. The buttons that appear on it vary, depending on which window is active.
These are the buttons that appear on the SpeedBar, accompanied by their descriptions that appear on the desktop status bar:

- ![Scissors icon](image)
  - Remove the selected text and put it in the Clipboard

- ![Clipboard icon](image)
  - Place a copy of the selected text in the Clipboard

- ![Clipboard icon](image)
  - Insert text from the Clipboard at the cursor position

- ![Undo icon](image)
  - Undo the previous editor action

- ![Folder icon](image)
  - Locate and open a file

- ![Save icon](image)
  - Save the file in the active Edit window

- ![Folder icon](image)
  - View the include files for project item

- ![Search bar](image)
  - Search for text

- ![Magnifying glass icon](image)
  - Repeat last Find or Replace operation

- ![Compiler icon](image)
  - Compile the selected file

- ![Update icon](image)
  - Bring target up-to-date

- ![Lightning bolt icon](image)
  - Make and run the current program

- ![Stopwatch icon](image)
  - Trace into statement

- ![Stopwatch icon](image)
  - Step over statement

- ![Lightning bolt icon](image)
  - Trace into instruction

- ![Stopwatch icon](image)
  - Step over instruction

- ![Question mark icon](image)
  - Access online help

- ![Folder icon](image)
  - Open the Project Manager window

- ![Settings icon](image)
  - Modify project wide settings and options
Some of the buttons on the SpeedBar are occasionally dimmed, just as some menu commands occasionally are. This means that, in the current context, the command the button represents is not available to you. For example, the Compile The Selected File button is dimmed if the selected file is not compilable (for example, if the selected file is a .DEF file).

From the keyboard, you can use a number of keyboard shortcuts (also known as hot keys) to access the menu bar, choose commands, or work within dialog boxes. You need to hold down Alt while pressing the highlighted letter when moving from an input box to a group of buttons or boxes. Here's a list of the keyboard shortcuts available:

<table>
<thead>
<tr>
<th>To accomplish this:</th>
<th>Do this:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display the menu, carry out the command, or select the button or menu choice</td>
<td>Press Alt plus the underlined letter of the command (in a dialog box, just press the underlined letter). For the File menu, you can press Alt by itself or F10.</td>
</tr>
<tr>
<td>Open the System menu</td>
<td>Press Alt+Spacebar.</td>
</tr>
<tr>
<td>Open the menu of the active window</td>
<td>Press Alt+ (the Alt key and the – key).</td>
</tr>
<tr>
<td>Carry out the command</td>
<td>Type the keystrokes next to a menu command.</td>
</tr>
</tbody>
</table>

For example, to cut selected text, press Alt+E T (for Edit | Cut) or you can just press Shift+Del, the shortcut.

There are also hot keys that perform functions without accessing any menus by means of a single keystroke. The following tables list the most-used Borland C++ hot keys.
### Table 2.1: General hot keys

<table>
<thead>
<tr>
<th>Hot key</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Help</td>
<td>Displays contextual help screen.</td>
</tr>
<tr>
<td>F10</td>
<td>none</td>
<td>Activates the menu bar.</td>
</tr>
<tr>
<td>Alt+F4</td>
<td>File</td>
<td>Exit</td>
</tr>
<tr>
<td>Alt+F5</td>
<td>none</td>
<td>Restores the desktop to its default size when minimized or maximized.</td>
</tr>
<tr>
<td>Alt+F7</td>
<td>none</td>
<td>Lets you move the desktop.</td>
</tr>
<tr>
<td>Alt+F8</td>
<td>none</td>
<td>Lets you size the desktop.</td>
</tr>
<tr>
<td>Alt+F9</td>
<td>none</td>
<td>Minimizes desktop.</td>
</tr>
<tr>
<td>Alt+F10</td>
<td>none</td>
<td>Maximizes desktop.</td>
</tr>
<tr>
<td>Alt+F11</td>
<td>none</td>
<td>Hides the desktop.</td>
</tr>
<tr>
<td>Alt+-</td>
<td>none</td>
<td>Opens the active window's system menu.</td>
</tr>
<tr>
<td>Alt+Spacebar</td>
<td>none</td>
<td>Opens the desktop's system menu.</td>
</tr>
<tr>
<td>Ctrl+F4</td>
<td>Window</td>
<td>Close</td>
</tr>
<tr>
<td>Ctrl+F6</td>
<td>Window</td>
<td>Next</td>
</tr>
</tbody>
</table>

### Table 2.2: Menu hot keys

<table>
<thead>
<tr>
<th>Hot key</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt+C</td>
<td>Compile menu</td>
<td>Takes you to the Compile menu.</td>
</tr>
<tr>
<td>Alt+D</td>
<td>Debug menu</td>
<td>Takes you to the Debug menu.</td>
</tr>
<tr>
<td>Alt+E</td>
<td>Edit menu</td>
<td>Takes you to the Edit menu.</td>
</tr>
<tr>
<td>Alt+F</td>
<td>File menu</td>
<td>Takes you to the File menu.</td>
</tr>
<tr>
<td>Alt+H</td>
<td>Help menu</td>
<td>Takes you to the Help menu.</td>
</tr>
<tr>
<td>Alt+P</td>
<td>Project menu</td>
<td>Takes you to the Project menu.</td>
</tr>
<tr>
<td>Alt+R</td>
<td>Run menu</td>
<td>Takes you to the Run menu.</td>
</tr>
<tr>
<td>Alt+S</td>
<td>Search menu</td>
<td>Takes you to the Search menu.</td>
</tr>
<tr>
<td>Alt+T</td>
<td>Tools menu</td>
<td>Takes you to the Tools menu.</td>
</tr>
<tr>
<td>Alt+W</td>
<td>Window menu</td>
<td>Takes you to the Window menu.</td>
</tr>
</tbody>
</table>
Table 2.3: Editing hot keys

<table>
<thead>
<tr>
<th>Hot key</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl+Ins</td>
<td>Edit</td>
<td>Copy</td>
</tr>
<tr>
<td>Shift+Del</td>
<td>Edit</td>
<td>Cut</td>
</tr>
<tr>
<td>Shift+Ins</td>
<td>Edit</td>
<td>Paste</td>
</tr>
<tr>
<td>Ctrl+Del</td>
<td>Edit</td>
<td>Clear</td>
</tr>
<tr>
<td>Alt+Bksp</td>
<td>Edit</td>
<td>Undo</td>
</tr>
<tr>
<td>Alt+Shift+Bksp</td>
<td>Edit</td>
<td>Redo</td>
</tr>
<tr>
<td>F3</td>
<td>Search</td>
<td>Search Again</td>
</tr>
</tbody>
</table>

Table 2.4: Online Help hot keys

<table>
<thead>
<tr>
<th>Hot key</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Help</td>
<td>Contents</td>
</tr>
<tr>
<td>F1 F1</td>
<td>none</td>
<td>Brings up Help on Help (just press F1 when you’re already in the help system).</td>
</tr>
<tr>
<td>Shift+F1</td>
<td>Help</td>
<td>Index</td>
</tr>
<tr>
<td>Ctrl+F1</td>
<td>Help</td>
<td>Topic Search</td>
</tr>
</tbody>
</table>

Table 2.5: Debugging/Running hot keys

<table>
<thead>
<tr>
<th>Hot key</th>
<th>Menu item</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>Edit</td>
<td>Local</td>
</tr>
<tr>
<td>Ctrl+F9</td>
<td>Run</td>
<td>Run</td>
</tr>
<tr>
<td>F4</td>
<td>Run</td>
<td>Go To Cursor</td>
</tr>
<tr>
<td>F7</td>
<td>Run</td>
<td>Trace Into</td>
</tr>
<tr>
<td>F8</td>
<td>Run</td>
<td>Step Over</td>
</tr>
<tr>
<td>F9</td>
<td>Compile</td>
<td>Make</td>
</tr>
</tbody>
</table>

Borland C++ windows

Most of what you see and do in the IDE happens in a window. A window is a screen area that you can open, close, move, resize, minimize, maximize, tile, and overlap.
You can have many windows open in the IDE, but only one window can be active at any time. Any command you choose or text you type generally applies only to the active window. (If you have the same file open in several windows, the action applies to the file everywhere that it's open).

You can spot the active window easily: It's the one with the colored bar at the top of it. If your windows are overlapping, the active window is usually the one on top of all the others (the foremost one). The only time the active window is not on top of all other open windows is when you have a window open that is not contained on the IDE desktop, such as the Settings notebook. The Settings notebook is always on top, even when it is not the active window.

There are several types of windows, but most of them have these things in common:

- A title bar
- A system menu button
- Scroll bars
- Window sizing buttons

The status line at the bottom of the desktop window also displays the current line and column numbers of the active edit window. If you've modified the file in the active window, the word "Modified" appears on the status line.

The system menu button of a window is the small box in the upper left corner. Double-click the system menu button to quickly close the window. You can also press Ctrl+F4, which closes the active window. The Inspector and Help windows are considered temporary; you can close them by pressing Esc.

The title bar, the topmost horizontal bar of a window, contains the name of the window. If the window contains a text file, the window name is the same as that of the open file. Otherwise, the window name indicates the function of the window. Click the title bar and move the mouse to drag the window to a new location. You can also double-click anywhere on the title bar to maximize the window, or, if it's already maximized, you can double-click anywhere on the title bar to restore the window to its normal size.

Window sizing buttons appear in the upper right corner of each window. The sizing buttons consist of a Minimize button, a Maximize button, and a Restore button. There are at most two of these buttons for each window, although there can be just one in some cases.
Pressing the Minimize button closes the window, and places an icon for it on the IDE desktop. You can then double-click the icon to restore it to its previous size.

Pressing the Maximize button opens the window to the full size of the Borland C++ desktop window. The Maximize button is then replaced by the Restore button. Pressing the Restore button returns the window to the same size as when you pressed the Maximize button. The Restore button is then replaced by the Maximize button.

Scroll bars are horizontal or vertical bars located on the bottom or right side of a window, respectively. You use these bars to scroll the contents of the window. Click the arrow at either end to scroll one line at a time. Keep the mouse button pressed to scroll continuously. You can click the shaded area to either side of the scroll box to scroll a page at a time. Finally, you can drag the scroll box to any spot on the bar to quickly move to a spot in the window relative to the position of the scroll box.

You can drag any corner or side of a window to make the window larger or smaller. Dragging a side lets you size the window in only one direction. For example, if you drag the bottom of the window, you can only make the window taller or shorter. But if you drag from the corner of a window, you can make it taller or shorter and wider or thinner.

Table 2.6 gives you a quick rundown of how to handle windows in Borland C++. Note that you don’t need to use the mouse to perform these actions—a keyboard works just fine.

<table>
<thead>
<tr>
<th>To accomplish this:</th>
<th>Do this:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open an edit window</td>
<td>Choose File</td>
</tr>
<tr>
<td>Open other windows</td>
<td>Click its desktop icon, or choose from the list in Window menu.</td>
</tr>
<tr>
<td>Close a window</td>
<td>Double-click the window’s system menu button, choose Close from the window’s menu, or press Ctrl+F4.</td>
</tr>
<tr>
<td>Activate a window</td>
<td>Click anywhere in the window, or choose the window from the list in the Window menu.</td>
</tr>
<tr>
<td>Move the active window</td>
<td>Drag its title bar, or choose Move from the window’s system menu and use the arrow keys to adjust the window position. Press Enter when done.</td>
</tr>
<tr>
<td>Resize the active window</td>
<td>Drag any corner or side of the window when the mouse pointer is a double-headed arrow, or choose Size from the window’s system menu and use the arrow keys to adjust the window size. Press Enter when done.</td>
</tr>
</tbody>
</table>
The status line

The status line appears at the bottom of the IDE desktop. It

- Tells you what the program is doing (for example, when an edit file is being saved, the status line displays Saving filename...).
- Offers one-line hints on any selected menu command and dialog box items.
- Indicates whether the file has been modified since the last time you saved it.
- Displays the current line and column position of the cursor when an edit window is active.

The status line changes as you switch windows or activities. When you’ve selected a menu title or command, the status line changes to display a one-line summary of the function of the selected item.

Dialog boxes

A menu command with an ellipsis (...) after it leads to a dialog box. Dialog boxes offer a convenient way to view and change multiple settings. When you’re making settings in dialog boxes, you work with five basic types of onscreen controls:

- Action buttons
- Radio buttons
- Check boxes
- Input boxes
- List boxes

Many dialog boxes have three standard buttons: OK, Cancel, and Help. If you choose OK, the choices in the dialog box are accepted; if you choose Cancel, nothing changes, no action takes place, and the dialog box is closed. Choosing Help opens a Help window containing information about your dialog box. Esc is always a keyboard shortcut for Cancel (even if no Cancel button appears).

To choose an item, click the dialog box button you want. If you want to use the keyboard, press Alt and the underlined letter of an item to activate it. For example, if the K in OK is underlined, Alt+K selects the OK button. Press Tab or Shift+Tab to move forward or backward from one item to another in a dialog box. Each element is highlighted when it becomes active.
You can select another button with Tab; press Enter to choose that button.

Radio buttons and check boxes

Most dialog boxes also have a default button that you can choose by simply pressing Enter. You can always tell which button is the default button because it's highlighted when the dialog box is first opened.

Radio buttons are like car radio buttons. They come in groups, and only one radio button in the group can be on at any one time. To choose a radio button, click it or its text. From the keyboard, select Alt and the highlighted letter, or press Tab until the group is highlighted and then use the arrow keys to choose a particular radio button. Press Tab or Shift+Tab again to leave the group with the new radio button chosen.

Check boxes differ from radio buttons in that you can have any number of check boxes checked at the same time. When you select a check box, a check mark appears in it to show you it’s on. An empty box indicates it’s off. To change the status of a check box, either click it or its text, press Tab until the check box is highlighted and then press Spacebar, or select Alt and the highlighted letter.

If several check boxes apply to a particular topic, they appear as a group. In that case, tabbing moves to the group. Once the group is selected, use the arrow keys to select the item you want, and then press Spacebar to check or uncheck it.

Input and list boxes

Input boxes let you type in text. Most basic text-editing keys work in the text box (for example, arrow keys, Home, End, and Ins). If you continue to type once you reach the end of the box, the contents automatically scroll right or left as necessary.

If an input box has a down-arrow icon (↓) to its right, you can display that box's history list or choice list. A history list is a list of the text you previously typed into this box. The Find box, for example, keeps track of and lists the text you searched for previously. Click the ↓ to display the list. To choose an item from the list, select it, then press Enter. You can also edit an entry in the history list, once it's in the input box. Press Esc to exit from the list without making a selection.

Many dialog boxes also contain a list box, which lets you scroll through and select from variable-length lists (often file names) without leaving a dialog box. If a blinking cursor appears in the list box and you know what you’re looking for, you can type the word (or the first few letters of the word) and Borland C++ searches the list for it.

To make a list box active, click it or choose the highlighted letter of the list title (or press Tab until it's highlighted). Once a list box is displayed, you
can use the scroll box to move through the list or press ↑ or ↓ from the keyboard.

Configuration and project files

IDE configuration files contain information about how you have the IDE environment configured. Project files contain all the information necessary to build a project, but don’t affect how you use the IDE.

The configuration file, TCCONFIG.TC, contains only environmental (or global) information, including
- Editor mode setting (such as autoindent, use tabs, and so on).
- Auto-save flags.

The configuration file is not required to build programs defined by a project. The project (.PRJ) file handles those details.

When you start a programming session, Borland C++ looks for TCCONFIG.TC first in the current directory and then in the directory that contains BC.EXE. If you delete TCCONFIG.TC, you can replace it with a default configuration file. The next time you start the IDE, choose the Project | Save menu command. Make sure the Environment box is checked and press OK.

Project files

The IDE places all information needed to build a program into a binary project file, a file with a .PRJ extension. Project files contain the settings for
- Compiler, linker, make, and librarian settings.
- Directory paths.
- The list of all files that make up the project.
- Special translators (such as Turbo Assembler).

In addition, the project file contains other general information on the project, such as compilation statistics (shown in the project window), and cached autodependency information.

.PRJ project files correspond to the .CFG configuration files that you supply to the command-line compiler (the default command-line compiler configuration file is TURBOC.CFG).

You can load project files in any of the following ways:
- When starting Borland C++ from the OS/2 command line, give the project name after the BC command; for example,
The project directory

- Specify a project file for a BC icon through the OS/2 Settings notebook. See page 16 for a description of how to set up project icons.
- If there is only one .PRJ file in the working directory when you start up the IDE, the IDE assumes that this directory is dedicated to this project and automatically loads the project file.
- To load a project from within the IDE, select Project | Open Project.

When a project file is loaded from a directory other than the current directory, the current directory is set to where the project is loaded from. This allows project items to be located relative to the current directory instead of by absolute paths, which allows projects to move from one drive to another or from one directory branch to another.

Desktop files

- Each project file can have an associated desktop file (prjname.DSK) that contains state information about the associated project. While none of its information is needed to build the project, all of the information is directly related to the project. The desktop file includes:
  - The context information for each window of the desktop (for example, your positions in the files or bookmarks).
  - The history lists for various input boxes (for example, search strings or file masks).
  - The layout of the windows on the desktop.
  - The contents of the Clipboard.
  - Watch expressions.
  - Breakpoints.

You don’t need to have the desktop file to use a project file. If you delete a desktop file, you can replace it by choosing the Project | Save menu command. Make sure the Desktop box is checked and press OK.

Default files

- When no project file is loaded, two default files serve as global placeholders for project- and state-related information: TCDEF.DPR and TCDEF.DSK files, collectively referred to as the default project.

These files are usually stored in the same directory as BC.EXE, and are created if they are not found. When you run the IDE from a directory without loading a project file, you get the desktop and settings from these files. These files are updated when you change any project-related settings (for example, compiler settings) or when your desktop changes (for example, the window layout).
When you start a new project, the settings from your previous project are in effect.

Because each project file has its own desktop file, changing to an existing project file causes the newly loaded project's desktop to be used, which can change your entire window layout. When you create a new project (by using Project | Open Project and typing in a new .PRJ file name), the new project's desktop inherits the previous desktop. When you select Project | Close Project, the default project is loaded and you get the default desktop and project settings.

Syntax highlighting helps you easily distinguish various parts of your code. Different syntax elements are highlighted in different colors for easy identification. For example, C and C++ keywords are highlighted a different color from identifiers. So when you look at your file in the editor, you can quickly pick out keywords from your variables and function names. Syntax items that are distinguished by syntax highlighting include the following:

- Breakpoint
- Character
- Comment
- CPU position
- Float
- Hex
- Identifier
- Illegal char
- Integer
- Octal
- Preprocessor
- Reserved word
- String
- Symbol
- Whitespace

Click the Syntax Highlighting check box in the Environment | Editor subsection of the Settings notebook to turn syntax highlighting on and off. To choose the colors, select the Environment | Syntax Hilite subsection of the Settings notebook.

To change the color of an element, follow these steps:

1. Select the element you want to change in the Element list box, or click a sample of that element in the code sample.
2. Select the colors you want in the Color dialog box.

   - To select a foreground color with your mouse, click the color in the FG box. To select the color with your keyboard, press Tab until the FG box is active. Use the arrow keys to move around the box.
To select a background color with your mouse, click the color in the BG box. To select the color with your keyboard, press Tab until the BG box is active. Use the arrow keys to move around the box.

As you select colors, you’ll see the results reflected in the sample code.

3. Close the Settings notebook.

Some basic tasks

This section contains descriptions of some basic tasks you can perform in the IDE, including compiling, linking, and debugging a program.

Compiling and linking programs

You can use the Borland C++ IDE to compile and link both single-file programs or multiple-file projects. There are a number of ways you can compile and/or link your application.

Making an application consists of the following steps:

1. Compile any source files that have been modified since they were last compiled, that include header files that have been modified, or that have not previously been compiled. This includes C and C++ files, assembly files (.ASM), resource script files (.RC), and any other text files that are processed into object or binary files.

2. Link the application if any of the link files (that is, object files, libraries, resource files, module-definition files (.DEF files), and so on) are newer than the existing executable file, or if there is no existing executable.

If you have a project file open, the project is built, regardless of what the current active window is. If you are compiling a single-file application without a project file, the source-file edit window must be the active window.

To make an application, do one of the following:

- Press F9.
- Choose the Compile | Make menu command.
- Press the Bring Target Up-to-date SpeedBar button.

Building an application is similar to making an application, except that all source files are compiled, regardless of whether they’ve been modified, and the application is linked. To build an application, choose the Compile | Build All menu choice.
You can choose to compile a single file as opposed to an entire application. This compiles the file in the active edit window or, if the Project window is the active window, compiles the file currently selected in the Project window. The Project Manager automatically uses the appropriate tool to compile a file. For example, the IDE uses the C++ compiler to compile a C++ file and the Resource Compiler to compile a resource script file (.RC file).

You cannot choose to compile if there are no windows open or if the Transcript window is the active window.

To compile a file, do one of the following:

- Choose the Compile | Compile menu command.
- Press the Compile The Selected File SpeedBar button.

You can link your object files into an executable without processing any source files, even if the source files have been modified since they were last compiled. To link an application, choose the Compile | Link menu command.

Once you have written and compiled your program, you might notice that it doesn’t produce the results you expected. This means you have a bug in your program. Borland C++ provides integrated debugging to let you track down program bugs, modify erroneous code, and rebuild your application, all without leaving the IDE. You can also use the standalone Turbo Debugger in much the same way as the IDE debugger. The IDE debugger is actually a functional subset of the standalone debugger. For more information on debugging, consult the Turbo Debugger User's Guide and the Turbo Debugger online help.

Before you can debug your application, it needs to contain debugging information. There are a number of settings that affect what debugging information is included in your application.

- There are four settings that pertain to debugging information located in the Compiler | Code Generation Options subsection of the Settings notebook:
  - Line Numbers Debug
  - Debug Info In OBJs
  - Browser Info In OBJs
  - Test Stack Overflow
These settings are described on page 64.

- The Out-of-line Inline Functions setting is located in the Compiler | C++ Options subsection of the Settings notebook. You should usually set this off unless you think there might be a problem specifically with inlining a function. This setting is described on page 66.

- The Include Debug Info settings is located in the Linker | Link Settings subsection of the Settings notebook. This controls whether debugging information is linked into the .EXE file. This setting is described on page 72.

You should set these settings to their appropriate values and rebuild your application. At a minimum, set the Debug Info In OBJs and Include Debug Info settings on. These let you debug your program from the source view or edit window.

After debugging you should set all debugging information settings off. This decreases the size of your object files and executables.

There are also settings you can use to customize how the integrated debugger acts during a debugging session.

- The Debugger Options section of the Settings notebook contains settings that affect:
  - What views the debugger opens in the event of a program exception.
  - What actions the debugger takes when displaying a message.
  - What syntax the debugger uses when evaluating user-input expressions.
  - How individual types of views behave.

The Debugger Options section settings are described on page 75.

- The Debug Source input box in the Directories section of the Settings notebook lets you specify the directory or directories where the debugger looks for the source code for libraries that do not belong to the open project (for example, container class libraries). The Debug Source input box is described on page 83.

- The SpeedBar options in the Environment | Desktop section let you specify how you want the SpeedBar displayed on the desktop. Among other things, the SpeedBar contains buttons that you can use for debugging. The SpeedBar options are described on page 85.

- The Environment | Syntax Highlighting subsection of the Settings notebook lets you configure the color of various syntax elements in IDE edit windows, including the CPU position and breakpoints. You can use this
to make the currently executing line and any breakpoints stand out for easy identification. Syntax highlighting is fully explained on page 30.

There are a number of ways you can observe data members from the IDE.

- You can inspect any data member accessible from the current scope using the Inspector view. To open the Inspector view, choose the Debug | Inspector menu command or the Edit Local | Inspect command. The Inspector view displays the variable name and its value. The Inspector view updates the variable value dynamically as it changes in the program.

- You can evaluate an expression using data members accessible from the current scope along with constants. The expression can contain a function call as long as the function used contains debugging information. To evaluate an expression, choose the Debug | Evaluator menu command or the Edit Local | Evaluate command. The Inspector view updates the variable value dynamically as it changes in the program. The Evaluator view updates the result of the expression dynamically as the variables in the expression change in the program.

- You can also display more than one data member or expression at a time using the Watch view. To open the Watch view, choose the Debug | Watch menu command or the Edit Local | Add Watch command. The Watch view displays each variable name and its value. The Watch view updates the variable values dynamically as they change in the program.

To find out where a particular bug is located, you need to be able to stop the execution of your application and test the values of program variables. There are a number of ways to stop program execution.

- Set a breakpoint. A breakpoint stops program execution at a specific point in the program code. To set a breakpoint, position the cursor on the line where you want execution to break. Press F2 or select Edit Local | Toggle Breakpoint. When you run the application, the debugger halts execution at the line you set the breakpoint on. You can then inspect the values of variables, register contents, and so on.

- Set a messagepoint. Messagepoints force the debugger to perform an action (usually stopping program execution) when the application receives a certain message or class of message from the Presentation Manager.

- Set a datapoint. Datapoints force the debugger to perform an action (usually stopping program execution) when a certain operation is performed on a data item or when the data item reaches a certain value.
- Set an exceptionpoint. Exceptionpoints force the debugger to perform an action (usually stopping program execution) when the application produces an exception.

To be able to stop program execution, you also need to be able to make your program execute. There are a number of ways you can execute your program.

- The simplest way to execute your program is simply to run it. This causes execution to begin at the current program counter (or at the beginning of the program if it hasn’t yet been run) and go until it either encounters an exception, meets the conditions for a breakpoint, datapoint, exceptionpoint, or messagepoint, or reaches the end of the program. To run your program do one of the following:
  - Press Ctrl+F9.
  - Choose the Run | Run menu command.
  - Press the Make And Run The Current Program SpeedBar button.

You can also start your program running and have it automatically stop at the current cursor position. To run your program to the current cursor position, do one of the following:
  - Press F4.
  - Choose the Run | Run To Cursor menu command or the Edit Local | Run To Cursor command.

You can execute your program incrementally, that is, step by step. Stepping over a statement executes the next line in your program. If that line is a function call, the function is executed as if it were a single statement. Execution stops at the line after the function call. To step over a statement, do one of the following:
  - Press F8.
  - Choose the Run | Step Over command or the Edit Local | Step Over command.
  - Press the Step Over Statement SpeedBar button.

Like stepping over a statement, tracing into a statement executes the next line in your program. But if that line is a function call, execution stops at the first line of the called function, letting you examine local variables and step through the function line by line. To trace into a statement, do one of the following:
  - Press F7.
  - Choose the Run | Trace Into command or the Edit Local | Trace Into command.
• Press the Trace Into Statement SpeedBar button.

There are also tracing and stepping equivalents for executing machine statements:

• Press the Step Over Instruction SpeedBar button to execute the next assembly statement. If the next statement is a call statement, the call is executed as if it were a single statement. Execution stops at the line after the call statement.

• Press the Trace Into Instruction SpeedBar button to execute the next assembly statement. But if the next statement is a call statement, execution stops at the first line of the called routine.
Menus and options reference

This chapter provides a reference to each menu option in the IDE. It is arranged in the order that the menus appear on the screen. For information on starting and exiting the IDE, using the IDE command-line options, and general information on how the IDE works, see Chapter 2.

*Alt+F4* Next to some of the menu option descriptions in this reference you’ll see keyboard shortcuts, or hot keys. For example, when you see *Alt+F4* beside a description, it means that is a hot key for that option.

**File menu**

*Alt+F* The File menu lets you open and create program files in edit windows. The menu also lets you save your changes, perform other file functions, and quit the IDE.

**New**

The File | New command lets you open a new edit window with the default name NONAME\textit{xx}.CPP (the \textit{xx} stands for a number from 00 to 63). These NONAME files are used as a temporary edit buffer; the IDE prompts you to name a NONAME file when you save it.

**Open**

The File | Open command displays a file-selection dialog box for you to select a program file to open in an edit window. The dialog box contains an input box, a drive selection box, a file type selection box, a file list, a directory list, and buttons labeled OK, Cancel, and Help. You can do any of these things:

- Type a full file name in the input box and press the Open button. Open loads the file into a new edit window.
- Type a file name with wildcards in the input box, which filters the file list to match your specifications.
- Press $\downarrow$ while the cursor is in the input box to choose a file specification from a history list of file specifications you’ve entered earlier.
- View the contents of different directories on your current drive by selecting a directory name from the directory list.
- View the contents of different drives by selecting a different drive name in the drive selection box.
- Close the dialog box by pressing the Cancel button or Esc.
- Get help regarding opening a file by pressing the Help button.

The input box lets you enter a file name explicitly or with standard OS/2 wildcards (* and ?) to filter the names appearing in the file list box. If you enter the entire name and press Enter, Borland C++ opens it. (If you enter a file name that Borland C++ can’t find, it automatically creates and opens a new file with that name.)

If you press Alt+↓ when the cursor is blinking in the input box, a history list drops down below the box. This list displays the last 15 file names or file name masks you’ve entered. Choose a name from the list by double-clicking it or selecting it with the arrow keys and pressing Enter.

Once you’ve typed in or selected the file you want, choose the Open button (choose Cancel if you change your mind). You can also just press Enter after the file is selected, or you can double-click the file name in the file list.

---

**Using the File list box**

In the file list and directory list, you can type any letter to search for a file or directory name that begins with that letter.

The file list box displays all file names in the current directory that match the specifications in the input box. Click in the file list box or press Tab until the first name in the file list box is highlighted. You can now press ↓ or ↑ to select a file name, and then press Enter to open it. You can also double-click any file name in the box to open it. You might have to scroll the box to see all the names.

---

**Save**

The File | Save command saves the file in the active edit window to disk (this menu item is disabled if there's no active edit window). If the file has a default name (NONAME00.CPP, or the like), the IDE opens the Save File As dialog box to let you rename and save it in a different directory or on a different drive. This dialog box is identical to the one opened for the Save As command, described next.

---

**Save As**

The File | Save As command lets you save the file in the active edit window under a different name, in a different directory, or on a different drive. Enter the new name, optionally with drive and directory, and click or choose OK. If the file is open in more than one window, then Borland C++ updates each of those windows with the new name.
Save All

The File | Save All command works just like the Save command except that it saves the contents of all modified files, not just the file in the active edit window. This command is disabled if no edit windows are open.

Print

The File | Print command lets you print the contents of the active edit window or the Transcript window. This command is disabled if the active window can’t be printed.

You can also print the contents of the Transcript window.

Exit

The File | Exit command exits the IDE and removes it from memory. If you have made any changes that you haven’t saved, the IDE asks you if you want to save them before exiting.

Closed File Listing

If you have opened files and then closed them, you’ll see the last five files listed at the bottom of the File menu. If you select the file name on the menu, Borland C++ opens the file. To reduce the clutter on the IDE desktop when you work with many files, you can close some, then open them again quickly using the list.

Edit menu

The Edit menu lets you cut, copy, and paste text in edit windows. If you make mistakes, you can undo changes and even reverse the changes you’ve just undone. You can also copy text from the Transcript window or the Help examples.

Before you can use most of the commands on this menu, you need to know about selecting text (because most editor actions apply to selected text). Selecting text means highlighting it. You can select text either with keyboard commands or with a mouse; the principle is the same even though the actions are different.

From the keyboard:

- Press Shift while pressing any key that moves the cursor.

See page 144 in Appendix B for additional text selection commands.
With a mouse:

- To select text with a mouse, drag the mouse pointer over the desired text. If you need to continue the selection past a window’s edge, just drag off the side and the window automatically scrolls.
- To select a single word, double-click it.
- To extend or reduce the selection, Shift-click anywhere in the document (that is, hold Shift and click).

Once you have selected text, the Cut, Copy, and Clear commands in the Edit menu become available.

The IDE uses the OS/2 Clipboard to hold text that you have cut or copied, so you can paste it elsewhere. The Clipboard works in close concert with the commands in the Edit menu.

Here's an explanation of each command in the Edit menu.

**Undo**

```
Alt+Backspace
```

Undo can undo groups of commands.

The Edit I Undo command restores the file in the current window to the way it was before the most-recent edit or cursor movement. If you continue to choose Undo, the editor continues to reverse actions until your file returns to the state it was in when you began your current editing session.

Undo inserts any characters you deleted, deletes any characters you inserted, replaces any characters you overwrote, and moves your cursor back to a prior position. If you undo a block operation, your file appears as it did before you executed the block operation. However, Undo does not change the contents of the OS/2 Clipboard. If you cut a section of text, then restore it by using Undo, the text still remains in the Clipboard.

Undo doesn’t change a setting that affects more than one window. For example, if you use the Ins key to change from Insert to Overwrite mode, then choose Undo, the editor won’t change back to Insert mode.

The Group Undo setting in the Environment I Editor subsection of the Settings notebook affects Undo and Redo. See page 86 for information on Group Undo.

**Redo**

```
Alt+Shift+Backspace
```

The Edit I Redo command reverses the effect of the most recent Undo command. The Redo command only has an effect immediately after an Undo command or after another Redo command. A series of Redo commands reverses the effects of a series of Undo commands.
The **Cut** command removes the selected text from your document and places the text in the Clipboard. You can then paste that text into any other document (or somewhere else in the same document) by choosing **Paste**, or by pressing `Shift+Ins`. The text remains in the Clipboard so that you can paste the same text many times.

The **Copy** command leaves the selected text intact but places a copy of it in the Clipboard. You can then paste the text into any other document by choosing **Paste**.

If the Transcript window is the active window when you select **Copy**, the entire contents of the window buffer (including any nonvisible portion) is copied to the Clipboard.

The **Paste** command inserts text from the Clipboard into the current edit window at the cursor position.

The **Clear** command removes the selected text but does not put it into the Clipboard. This means you cannot paste the text as you could if you had chosen Cut or Copy. The cleared text is not retrievable unless you use the **Undo** command. Clear is useful if you want to delete text, but you don't want to overwrite text being held in the Clipboard.

---

**Search menu**

**Find**

The **Find** command displays the Find Text dialog box, which lets you type in the text you want to search for and set options that affect the search. There is also a SpeedBar icon for Search.

Check the **Case Sensitive** box if you want the IDE to differentiate uppercase from lowercase.

Check the **Whole Words Only** box if you want the IDE to search for words only (that is, the string must have punctuation or space characters on both sides).
Check the Regular Expression box if you want the IDE to recognize GREP-like wildcards in the search string. The wildcards are ^, $, ., *, +, [, \{, \}, \i (0 <= i <= 9), (x), \<, \>, \|, x?, \t, \xhh, \dddd, and \c. Here's what they mean:

<table>
<thead>
<tr>
<th>Wildcards</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>A circumflex at the start of the string matches the start of a line.</td>
</tr>
<tr>
<td>$</td>
<td>A dollar sign at the end of the expression matches the end of a line.</td>
</tr>
<tr>
<td>.</td>
<td>A period matches any character.</td>
</tr>
<tr>
<td>*</td>
<td>A character followed by an asterisk matches any number of occurrences (including zero) of that character. For example, bo* matches b, bo, boo, booo, and so on.</td>
</tr>
<tr>
<td>+</td>
<td>A character followed by a plus sign matches any number of occurrences (but not zero) of that character. For example, bo+ matches bo, boo, booo, and so on, but not b.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Characters in brackets match any one character that appears in the brackets but no others. For example [bot] matches b, o, or t.</td>
</tr>
<tr>
<td>[ ]</td>
<td>A circumflex at the start of the string in brackets means not. Hence, [*bot] matches any characters except b, o, or t.</td>
</tr>
<tr>
<td>[- ]</td>
<td>A hyphen within the brackets signifies a range of characters. For example, [b-o] matches any character from b through o.</td>
</tr>
<tr>
<td>\</td>
<td>A backslash before a wildcard character tells Borland C++ to treat that character literally, not as a wildcard. For example, ^ matches ^ and does not look for the start of a line.</td>
</tr>
<tr>
<td>{x}</td>
<td>Enclosing an expression in curly braces &quot;tokenizes&quot; it, which lets you use a string that matches the expression in your replacement string. Tokens are named from left to right, starting at 0. For example, suppose you want to search for the expression in[<em>]de and change the first two letters to pre. You would search for the expression in[</em>]de, and replace it with the expression pre\0de. Thus the word include would become preclude. Because the * inside the curly braces matches with the string clu in include, \0 represents clu.</td>
</tr>
<tr>
<td>(x)</td>
<td>You can use parentheses to group together regular expressions, much like you do in a language statement. See the explanation of the</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt; means that the expression must be located at the beginning of a word. For example, &lt;keep matches keeper, but not bookkeeper.</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt; means that the expression must be located at the end of a word. For example, &gt;keep matches barkeep, but not keeper.</td>
</tr>
</tbody>
</table>

You can use \< and \> to force a word search. For example, \<k*pl> matches keep, but not keeper or barkeep.

| \   | You can use this to match one of a number of sequences in an expression. For example, the expression (b|k|s)een would match the words been, keen, and seen. |
Table 3.1: Search-string wildcards (continued)

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c?</code></td>
<td>This tells Borland C++ to search for one or no appearances of c (which can be any character). For example, the string <code>?ama</code> matches both <code>ama</code> and <code>lama</code>.</td>
</tr>
<tr>
<td>\t</td>
<td>This matches a tab character.</td>
</tr>
<tr>
<td>\xhh</td>
<td>This matches a character with the ASCII value of hexadecimal <code>hh</code>.</td>
</tr>
<tr>
<td>\ddd</td>
<td>This matches a character with the ASCII value of decimal <code>ddd</code>. For example, <code>/d64</code> matches the character <code>@</code>.</td>
</tr>
<tr>
<td>\c</td>
<td>Specifying this anywhere in a regular expression tells the editor to place the cursor in the string when <code>\c</code> is placed. For example, if you search for the string <code>alcbc</code>, the editor would place the cursor after the <code>a</code>.</td>
</tr>
</tbody>
</table>

Enter the string in the input box and choose OK to begin the search, or choose Cancel to forget it. If you want to enter a string that you searched for previously, press Alt+\downarrow to show a history list to choose from.

You can also pick up the word that your cursor is currently on in the edit window and use it in the Find Text box by simply invoking Find from the Search menu.

Choose from the Direction radio buttons to decide which direction you want the IDE to search—starting from the origin (which you can set with the Origin radio buttons).

Choose from the Scope buttons to determine how much of the file to search in. You can search the entire file (Global) or only the text you've selected.

Choose from the Origin buttons to determine where the search begins. When Entire Scope is chosen, the Direction radio buttons determine whether the search starts at the beginning or the end of the scope. You choose the range of scope you want with the Scope radio buttons.

The Search | Replace command displays the Replace Text dialog box that lets you type in text you want to search for and text you want to replace it with. The Replace Text dialog box contains several radio buttons and check boxes—many of which are identical to the Find Text dialog box, discussed previously. An additional checkbox, Prompt on Replace, controls whether you're prompted for each change.

Enter the search string and the replacement string in the input boxes and choose OK or Change All to begin the search, Cancel to forget it, or Help to open online Help for the Replace box. If you want to enter a string you used previously, press Alt+\downarrow to show a history list to choose from.

If the IDE finds the specified text and Prompt on Replace is on, it asks you if you want to make the replacement. If you choose OK, it finds and
replaces only the first instance of the search item. If you choose Change All, it replaces all occurrences found, as defined by Direction, Scope, and Origin.

Search Again

The Search | Search Again command repeats the last Find or Replace command. All settings you made in the last dialog box used (Find or Replace) remain in effect when you choose Search Again. There is also a SpeedBar icon for Search Again.

Go to Line Number

The Search | Go to Line Number command prompts you for the line number you want to find.

Run menu

The Run menu's commands run your program and let you specify optional command-line arguments for your program.

Run

The Run | Run command runs your program, using any arguments you pass to it with the Run | Arguments command. If the source code has been modified since the last compilation, it also invokes the Project Manager to recompile and link your program. The Project Manager is a program-building tool incorporated into the IDE; see Chapter 5, "Managing multi-file projects," for more on this feature.

If you don't want to debug your program in Borland C++, you can compile and link it with the debugging settings turned off (which makes your program link faster) in the Code Generation Options subsection of the Compiler section in the Settings notebook. If you compile your program with the debugging settings on, the resulting executable code contains debugging information that affects the behavior of the Run | Run command in the following ways.

Using the same source code

If you have not modified your source code since the last compilation, the Run | Run command causes your program to run to the next breakpoint, or to the end if no breakpoints have been set.

Using modified source code

If you have modified your source code since the last compilation, and you're already stepping through your program using the integrated debugger, Run | Run prompts you to ask whether you want to rebuild your program.
If you answer yes, the Project Manager recompiles and links your program, and sets it to run from the beginning.

If you answer no, your program runs to the next breakpoint or to the end if no breakpoints are set.

Alternatively, if you have modified your source code since the last compilation but you’re not in an active debugging session, the Project Manager recompiles your program and sets it to run from the beginning.

The Run | Step Over command causes Borland C++ to execute the next line in your program. If that line is a function call, the function is executed as if it were a single statement. Execution stops at the line after the function call.

The Run | Trace Into command causes Borland C++ to execute the next line in your program. If that line is a function call, execution stops at the first line of the called function, letting you examine local variables and step through the function line by line.

The Run | Run To Cursor command begins program execution from the current program counter and runs it until it encounters the line at which the cursor is positioned. The debugger halts execution, letting you test the state of your program.

The Run | Reset command resets the program counter to the beginning of your program, clearing all allocated memory. After resetting your program, it is in essentially the same state it was in before you began running it.

The Run | Run Arguments command lets you give your running programs command-line arguments exactly as if you had typed them on the OS/2 command line or specified them in program’s Setting notebook. OS/2 redirection commands such as < or > are ignored.

When you choose this command, a dialog box appears with a single input box. You only need to enter the arguments here, not the program name. Arguments take effect when your program starts.

If you are already debugging and want to change the arguments, select Run | Reset and Run | Run to start the program with the new arguments.
Compile menu

Use the commands on the Compile menu to compile the program in the active window or to make or build your project. To use the Compile, Make, Build, and Link commands, you must have a file open in an active edit window or a project defined.

**Compile**

The Compile | Compile command compiles the file in the active edit window. If the Project or Transcript window is active, Compile | Compile compiles the highlighted file.

When the compiler is compiling, the Transcript window opens up to display the compilation progress and results. If any errors or warnings occurred, they are displayed in the Transcript window.

**Make**

The Compile | Make command invokes the Project Manager to compile and link your source code to the target executable or library.

Compile | Make rebuilds only the files that aren't current.

The target file name listed is derived from one of two names in the following order:

- Project file (.PRJ) specified with the Project | Open Project command.
- Name of the file in the active edit window. If no project is defined, you’ll get the default project defined by the file TCDEF.DPR.

The extension given to the output file depends on what type of application the file is.

**Link**

The Compile | Link command takes the files defined in the current project file or the defaults and links them.

**Build All**

This command is similar to Compile | Make except that it rebuilds all the files in the project whether or not they are current. It performs the following steps:

1. Deletes the appropriate precompiled header (.CSM) file, if it exists.
2. Deletes any cached autodependency information in the project.
3. Sets the date and time of all the project’s .OBJ files to zero.
4. Does a make.
If you abort a Build All command by choosing the Compile | Break menu command, pressing Ctrl+Break, or getting errors that stop the build, you can pick up where it left off by choosing Compile | Make.

Choosing the Break command while building a program terminates the build process.

Debug menu

The commands on the Debug menu control all the features of the integrated debugger. You can access these features through windows known as views. Each Debug menu command opens a view that lets you perform such tasks as setting breakpoints, viewing the disassembled program code, and evaluating expressions.

Each view provides a special local menu for debugging; this feature is not available directly through the IDE menus. You can access the local menu for a view by right-clicking anywhere in the view, or by pressing Shift+F10. For an explanation of the features of an individual view, press Ctrl+F1 or choose the Help | Topic Search menu command while the view is active. For more information on local menus, see page 19.

Debugging can be affected by the settings in the following sections of the Settings notebook:

- The settings in the Compiler | Code Generation Options subsection affect what types of debugging information the compiler includes in generated object code modules.
- The Include Debug Info setting in the Linker | Link Settings subsection affects whether debugging information is linked into your executable module.
- The Debugger section affects how the Borland C++ integrated debugger performs in the IDE environment.
- The Debug Source box in the Directories section specifies the directories where the Borland C++ integrated debugger looks for the source code for libraries that do not belong to the open project (for example, container class libraries).
- The Environment | Syntax Highlighting subsection of the Settings notebook lets you configure the color of various syntax elements in IDE edit windows, including the CPU position and breakpoints. This lets you make the currently executing line and any breakpoints stand out for easy identification.
The Breakpoints command opens the Breakpoint view. You can use the Breakpoint view to set, modify, and delete program breakpoints. Breakpoints are places in your program where the debugger performs a prescribed action (such as breaking execution or evaluating an expression), letting you inspect the state of program variables and objects.

You can also set and clear a breakpoint at the current cursor position by pressing F2 or by choosing the Edit Local | Toggle Breakpoint command. If there is already a breakpoint on the current line, either action removes the breakpoint. If there is not a breakpoint, either action sets a breakpoint on the current line.

The Messagepoints command opens the Messagepoint view. You can use the Messagepoint view to set, modify, and delete messagepoints. Messagepoints force the debugger to perform an action when the application receives a certain message or class of messages from the Presentation Manager, letting you inspect the state of your application.

The Datapoints command opens the Datapoint view. You can use the Datapoint view to set, modify, and delete datapoints. Datapoints force the debugger to perform an action when a certain operation is performed on a data item or when the data item reaches a certain value.

The Exceptionpoints command opens the Exceptionpoint view. You can use the Exceptionpoint view to set, modify, and delete exceptionpoints. Exceptionpoints force the debugger to perform an action when the application produces an exception.

The Source command opens a Source view. A Source view in the IDE is the same as an editor window.

The Disassembly command opens a Disassembly view. This displays the disassembled program code for your application.

The Variable command opens a Variable view. This displays the names and values of all variables local to the current function.

The Call Stack command opens a Call Stack view. This displays the current state of the program call stack.
The Watch command opens a Watch view. You can use the Watch view to monitor the values of multiple variables.

The Evaluator command opens an Evaluator view. The Evaluator view lets you evaluate an expression.

The Inspector command opens an Inspector view. You can use the Inspector view to display the contents of a variable, follow pointers, change the value of a variable, and so on.

The Thread command opens a Thread view. The Thread view displays the current function, process ID, thread ID, and status of each of your application's threads.

The Memory command opens a Memory view. You can use the Memory view to display the contents of a certain area of memory, change the contents, search for a string in memory, and so on.

The Registers command opens a Register view. The Register view lets you view the contents of the CPU registers. You can also modify the contents of the registers.

The Numeric Processor command opens a Numeric Processor view. The Numeric Processor view lets you see the contents of the numeric processor registers. You can also modify the contents of the numeric processor registers.

The Heap command opens the Heap view. The Heap view lets you display and modify the program heap.

The Hide Windows command hides all debugger views that are currently open.

The Show Windows command opens all debugger views that have been hidden using the Hide Windows command.
The Tools menu contains a number of customizable commands that you can use to perform functions that the IDE does not provide itself. The Tools menu also provides facilities to track error messages displayed in the Transcript window.

The Tools | View Transcript command brings the Transcript window to the front of the desktop. If the Transcript window was previously closed it is reopened, then brought to the front.

The Tools | Previous Error command moves the cursor to the location of the previous error or warning message. This command is available only if there are messages in the Transcript window that have associated line numbers.

The Tools | Next Error command moves the cursor to the location of the next error or warning message. This command is available only if there are messages in the Transcript window that have associated line numbers.

The Compile | Remove Messages command removes all messages from the Transcript window.

At the bottom of the Tools menu are the names of various programs you can execute from the IDE. You can use the Transfer section of the Settings notebook to customize the programs listed here.

A program that appears here on the Tools menu can be run directly from the IDE. You can install or delete programs here through the Transfer section of the Settings notebook. To run one of these programs, choose its name from the Tools menu.

If you have more than one program installed with the same shortcut letter on this menu, the first program listed with that shortcut is selected. You can select the second item by clicking it or by using the arrow keys to move to it and then pressing Enter.

To provide error tracking for tools you place on the Tools menu, you can write a transfer filter that directs the output from the tool to the IDE Transcript window. Borland provides transfer filters for a number of tools that are contained on the default Tools menu: GREP2MSG.EXE for GREP, IMPL2MSG.EXE for IMPLIB, BRCC2MSG.EXE for the Resource Compiler, RC2MSG.EXE for the Resource Binder, and TASM2MSG.EXE for Turbo Assembler (TASM). We've included the source code for these filters so you
can write your own filters for other transfer programs you install. This is explained in more detail in the online file UTIL.DOC.

### Project menu

Alt+P  The Project menu contains all the project management commands to

- Create a project.
- Add or delete files from your project.
- View included files for a specific file in the project.
- Set local options for a single file within a project.

The Open Project command displays the Open Project File dialog box, which lets you select and load a project or create a new project by typing in a name.

This dialog box lets you select a file name similar to the File | Open dialog box, discussed on page 37. The IDE uses the file you select as a project file, which is a file that contains all the information needed to build your project's executable. Borland C++ uses the project name when it creates the .EXE, .DLL, or .LIB file and .MAP file. A typical project file has the extension .PRJ.

Choose Project | Close Project when you want to remove your project and return to the default project.

The Project | View Project command brings the Project window to the front of the desktop. If the Project window was previously closed it is reopened, then brought to the front.

The Project | View Settings command opens the Settings notebook for the current project. You can use the Settings notebook to view and modify various settings for your projects. See Chapter 4, "Settings notebook" for more information about the Settings notebook.

Choose Project | Add Item when you want to add a file to the project's file list. This brings up the Add to Project List dialog box.

This dialog box is set up much like the Open a File dialog box (File | Open). Choosing the Add button puts the currently highlighted file in the Files list into the Project window. The chosen file is added to the Project window.
File list immediately after the highlight bar in the Project window. The highlight bar is advanced each time a file is added (when the Project Window is active, you can press Ins to add a file).

Choose Project I Delete Item when you want to delete the highlighted file in the Project window. When the Project window is active, you can press Del to delete a file.

The following command-line options are not supported: c, Efilename, e, Ipathname, L, lx, M, Q, y.

The Local Options command opens the Override Options dialog box. This dialog box lets you include command-line override options for a particular project-file module. It also lets you give a specific path and name for the object file and lets you choose a translator for the module.

Any program you installed in the Modify/New Transfer Item dialog box with the Translator box checked appears in the list of Project File Translators (see page 90 for information on the Modify/New Transfer Item dialog box).

Check the Exclude Debug Information setting to prevent debug information included in the module you’ve selected from going into the .EXE.

Use this switch on already debugged modules of large programs. You can change which modules have debug information simply by checking this box and then re-linking (no compiling is required).

Check the Exclude from Link option if you don’t want this module linked in.

Choose Project I Include Files to display the Include Files dialog box or, if you’re in the Project window, press the Spacebar. If you haven’t built your project yet, the Project I Include Files command is disabled.

The Include Files dialog box displays a list of all the include files included by the selected project item, including files included by other include files. You can scroll through the list of files displayed. Select the file you want to view and press Enter. Borland C++ then opens the file in an edit window.

The Generate Makefile command produces a makefile that you can use with the Borland Make utility to generate your application from the OS/2 command line. The makefile is given the name PRJ_NAME.MAK.
Save

The Project | Save opens the Save Options dialog box, which lets you save the environment, desktop, project, or any combination of these three. To set any one of these on or off, click the corresponding radio button. To save the desired attributes, press OK. To close the Save Options dialog box without saving anything, press Cancel. These are saved only if they've been changed since the last time they were saved.

Window menu

Alt+W

The Window menu contains window management commands. Most of the windows you open from this menu have all the standard window elements like scroll bars, a system menu button, and a Minimize and Maximize button. Refer to page 23 for information on these elements and how to use them.

Tile

Choose Window | Tile to tile all open windows on the IDE desktop, including your edit windows, the Project window, and the Transcript window.

Cascade

Choose Window | Cascade to stack all open windows on the IDE desktop, including your edit windows, the Project window, and the Transcript window.

Arrange Icons

Choosing Window | Arrange Icons rearranges any icons on the IDE desktop so they are evenly spaced, beginning at the lower left corner of the desktop window.

Close All

Close All closes all open windows on the Borland C++ desktop.

Open Windows Listing

At the bottom of the Window menu is a list of windows open on the Borland C++ desktop. If there are more than ten open windows, the last choice on the menu is More. Choosing this replaces the list of windows currently on the menu with other open windows. You can use this to page through all the open windows on your desktop. Choosing a window makes that window the active one.
Help menu

The Help menu gives you access to online Help in a special window. There is help information on virtually all aspects of the IDE and Borland C++. (Also, one-line menu and dialog box hints appear on the status line whenever you select a command.)

To open the Help window in Borland C++, do one of these actions:

- **Press F1** at any time (including from any dialog box or when any menu command is selected). Every item in a dialog box has its own context-sensitive help.
- When an edit window is active and the cursor is positioned on a word, press Ctrl+F1 to get language help on that word.
- Click Help whenever it appears in a dialog box.
- While in the IDE desktop, press Alt+H to go to the Help menu.

To close the Help window, press Esc, double-click the system menu button, or press Ctrl+F4. You can keep the Help window onscreen while you work in another window unless you opened the Help window from a dialog box or pressed F1 when a menu command was selected.

Help screens often contain links (highlighted text) that you can choose to get more information. Press Tab to move to any link; press Enter to get more detailed help about the highlighted link. As an alternative, use the arrow keys to move the cursor to the highlighted link and press Enter. With a mouse, you can double-click any link to open the help text for that item.

You can also move the cursor to any open window on the Borland C++ desktop (other than a Help window) and press Ctrl+F1 on any word to get help. If the word is not found, the Help system performs an incremental search through all its files and the closest match displayed.

When the Help window is active, you can copy from the window and paste that text into an edit window. You do this just the same as you would in an edit window: Select the text first, choose Edit | Copy, move to an edit window, then choose Edit | Paste.

You can copy example code from Help topics and compile it or use it in the program you are writing. There are two ways to do this:

- To copy the contents of a Help window to the OS/2 Clipboard,
  
    1. Make the Help window from which you want to copy text active by clicking anywhere in the window.
2. Select Copy from the Services menu of the Help window, or press Ctrl+Ins. The contents of the Help window are now stored in the OS/2 Clipboard.

3. You can paste the selected text into an editor window by pressing Shift+Ins while the window is active.

To copy the contents of a Help window to a file,

1. Make the Help window you want to copy from active by clicking anywhere in the window.

2. Select Copy To File from the Services menu of the Help window, or press Ctrl+F. The contents of the Help window are now stored in a file named TEXT.TMP in the current directory. You can edit this file using any text editor.

TEXT.TMP is overwritten every time you use the Copy To File command, so you should probably rename the file before you continue.

If you want to copy more than one topic into a file, follow the same procedure, but use the Append To File command instead of Copy To File.

For both of these methods, you can copy the contents of an entire window only. You might need to edit the copied text to remove any additional text that was copied along with the example code.

Borland C++ for OS/2 uses the OS/2 Help system. If you know how to use Help in other OS/2 applications, you'll know how to get help in Borland C++.

The Help | Contents command opens the Help window with the main table of contents displayed. From this window, you can branch to any other part of the help system.

You can get help on Help by pressing F1 when the Help window is active. You can also reach this screen by clicking on the status line.

The Help | Index command displays an index of the Borland C++ Help system, letting you quickly locate any subject for which you need help.

You can scroll or search the list by pressing letters from the keyboard. When you type a letter, the cursor jumps to the first index heading that starts with that letter. Press it again, and the cursor goes on to the first subheading that starts with that letter. If there is no subheading that starts with that letter, the cursor goes to the first heading that starts with that letter.
When you find an index entry that interests you, choose it by placing the cursor on it and pressing Enter (you can also double-click it).

**Topic Search**  
*Ctrl+F1*

The Help | Topic Search command displays language help on the currently selected item.

To get language help, position the cursor on an item in an edit window and choose Topic Search. You can get help on things like function names (printf, for example), header files, reserved words, and so on. If an item is not in the help system, the help index displays the No Help Available window.

**Essentials**

The Help | Essentials command contains a description of the differences between the Borland C++ IDE for DOS and Windows and the IDE for OS/2. It also contains a list of online files. If you have used Borland C++ under one of these other operating systems, you can use this facility to help find your way around easily.

**Language Reference**

The Help | Language Reference command provides an alphabetical reference to the Borland C++ standard run-time library, including the Borland C++ class libraries.

**Error Messages**

The Help | Error Messages command provides a complete description of compile- and run-time error messages, with some suggestions for fixing the error condition.

**Tasks**

The Help | Tasks command contains descriptions of how to perform a number of common tasks using the Borland C++ IDE, including compiling and linking files.

**Menus**

The Help | Menus command provides an online summary of all IDE menu choices.

**Keyboard**

The Help | Keyboard command provides an online summary of all IDE keyboard shortcuts.

**Using Help**  
*F1+F1*

The Help | Using Help command opens up a text screen that explains how to use the Borland C++ help system. If you're already in help, you can bring up this screen by pressing F1.
When you choose this command, a dialog box appears that shows you copyright and version information for Borland C++ for OS/2. Click OK or press *Enter* to close the box.
Settings notebook

The Settings notebook lets you view and change various settings in Borland C++ by paging through a graphic notebook containing several sections and subsections. Each section and subsection contains groups of related settings. You can use these settings to customize the behavior of the IDE editor, compiler, linker, and so on.

Using the Settings notebook

To open the Settings notebook, choose Project | View Settings from the menu bar.

Unlike other windows in Borland C++ for OS/2, you cannot minimize the Settings notebook. You can move the notebook so that you can see the IDE desktop as you change settings in the notebook.

You can close the Settings notebook in four ways:

- Double-click the system menu button.
- Click the system menu button and choose Close.
- Press Alt+Spacebar and choose Close.
- Press Alt+F4.

Sections in the Settings notebook are graphically represented by divider tabs on the right edge of the notebook. Subsections are denoted by divider tabs on the bottom edge of the notebook. The subsection tabs change according to which section you’re currently viewing. Because some sections don’t have any subsections, there are no tabs along the bottom of the notebook in these sections.
You can move from one section or subsection to another by clicking on the divider tabs on the right and bottom edges of the notebook.

You can go to any section or subsection in the notebook by simply clicking on the proper tab. The current section (and subsection, if applicable) is highlighted and raised to the top of the notebook.

You can move from page to page by clicking on the arrows in the lower-right corner of the notebook page. These move you back one page if you click the left arrow, or forward one page if you click the right arrow. If you run out of pages in your current section or subsection, you go to the next one. Some sections also have double arrows below the notebook; these let you move through the subsection tabs.

The notebook is made up of the following sections:

- **Compiler** contains settings that affect the behavior of the IDE compiler. These settings are organized into the following subsections:
  - Code Generation Options contains settings that directly affect how the compiler generates code.
  - C++ Options contains settings that specify how the compiler should handle C++ code.
  - Optimizations contains settings that let you set certain optimizations on and off.
  - Source Options contains settings that let you specify what standard your code complies with, Borland C++, ANSI C, or Kernighan & Ritchie.
  - Messages lets you control the messages output by the IDE compiler.
  - Names lets you specify the names of code segments.

- **Make** contains settings that affect the functioning of the IDE make process.

- **Target** contains settings that specify the type of executable the IDE produces.

- **Linker** contains settings that specify how an application should be linked, including which link libraries to use and whether to include debugging information. These settings are organized into three subsections:
  - Link Settings lets you specify how the linker links your project.
  - Link Libraries lets you specify which libraries you want to link with your application.
  - Link Warnings lets you control the warnings output by the IDE linker.
Librarian contains settings that affect the behavior of the built-in librarian.

Debugger Options contains settings that affect the behavior of the integrated debugger. These settings are organized into the following subsections:

- Debugger Options contains settings that let you specify how the Borland C++ integrated debugger performs in the IDE environment.
- Disassembly View Local Options contains settings that let you customize the appearance of the integrated debugger disassembly view.
- Variables View Local Options contains settings that let you customize the appearance of the integrated debugger variables view.
- Call Stack View Local Options contains settings that let you customize the appearance of the integrated debugger call stack view.
- Watch View Local Options contains settings that let you customize the appearance of the integrated debugger watch view.
- Evaluator View Local Options contains settings that let you customize the appearance of the integrated debugger evaluator view.
- Inspector View Local Options contains settings that let you customize the appearance of the integrated debugger inspector view.
- Memory View Local Options contains settings that let you customize the appearance of the integrated debugger memory view.
- Register View Local Options contains settings that let you customize the appearance of the integrated debugger register view.
- File And Numeric View Local Options contains settings that let you customize the appearance of the integrated debugger file and numeric processor views.

Directories contains settings that specify paths the IDE uses to locate header files, library files, source code, and so on.

Environment contains settings that let you modify the "look and feel" of the IDE. These settings are organized into the following subsections:

- Preferences contains settings that let you specify the general behavior of the IDE environment.
- Desktop lets you specify what portions of the desktop you want saved and where on the desktop you want to position the SpeedBar.
- Editor contains settings that let you tailor the behavior of the IDE editor.
- Fonts lets you specify the font size and style Borland C++ uses in its windows.
• Syntax Hilite lets you customize the colors and styles used to denote syntax elements when syntax highlighting is on.

• Transfer contains a list of programs that are included on the transfer item section of the Tools menu.

When you first view the Settings notebook, certain settings are already selected. These are the default settings, which Borland C++ uses if you do not make any changes.

You can change the default settings by making the desired changes and selecting the Project | Save menu command. When the Save Options dialog box opens, make sure the Project box is checked, and press OK.

You can also set Borland C++ to automatically save your settings at the end of each programming session. Open the Settings notebook, go to the Environment section, and turn to the Preferences subsection. Turn on the Project box in the Autosave section. Now any changes you make to the settings are automatically saved when you exit from Borland C++ or when you exit from a project.

Specific sections and subsections in the Settings notebook are referenced here by the same notation that is used for menu choices. For example, Compiler | Code Generation Options refers to the Code Generation Options subsection of the Compiler section. Within this chapter, we do not specify that these choices are in the Settings notebook. But we do specify when a choice is a selection from a menu bar.

Unlike the DOS and Windows versions of Borland C++, where changes in IDE settings do not take effect until you leave the Option menus, changes made in the Settings notebook take effect as soon as you make them.

With the exception of the Transfer section, each page in the notebook has an Undo button and a Default button. Pressing the Undo button on a page restores that page to the state it was in when you opened the Settings notebook. Pressing the Default button on a page restores that page to the Borland-supplied default state.

Compiler section

The Compiler section lets you modify settings that affect

• Compiler code generation.
• How C++ files are compiled.
• Which optimizations to use.
The Code Generation Options subsection (labeled Code Gen on the subsection tab) contains settings that let you specify how the compiler generates code, with settings for including debugging information in object files, function calling conventions, and miscellaneous code settings.

The Options settings box contains a number of settings that you can use to tailor object-code generation.

- **Treat Enums As Ints** forces the compiler to always allocate a four-byte `int` for variables of type `enum`. When this setting is off, the compiler allocates an unsigned or signed byte if the minimum and maximum values of the enumeration are both within the range of 0 to 255 or -128 to 127, respectively, or an unsigned or signed `short` if the minimum and maximum values of the enumeration are both within the range of 0 to 65,535 or -32,768 to 32,767, respectively. Treat Enums As Ints is off by default.

- **Word Alignment** tells Borland C++ to align noncharacter data (within `structs` and `unions` only) at 32-bit word (4-byte) boundaries. When this setting is off, Borland C++ uses byte-aligning, where data (again, within `structs` and `unions` only) can be aligned at either odd or even addresses, depending on the next available address.

Word Alignment increases the speed at which 80x86 processors fetch and store data.

- **Unsigned Characters** tells Borland C++ to treat all `char` declarations as if they were type `unsigned char`. When unchecked, `chars` are treated as `signed chars`, unless the `unsigned` keyword is specified in the source code.

- **Merge Duplicate Strings** tells Borland C++ to merge two strings when one matches another. This produces smaller programs, but can introduce bugs if you modify one string.

- **Precompiled Headers** tells the IDE to generate and use precompiled headers. Precompiled headers can dramatically increase compilation speeds, though they require a considerable amount of disk space. When this setting is off (the default), the IDE neither generates nor uses precompiled headers. Precompiled headers are saved in the file `PRJ_NAME.CSM`.

---

See Appendix C for more on precompiled headers.
Compiler|Code Generation Options

- Generate Assembler Source tells the IDE to produce an .ASM assembly language source file as its output, rather than an .OBJ object module. Because the compiler does not produce an .OBJ file when this setting is on, you should use the Compile | Compile menu choice so that the IDE does not try to invoke the linker. The linker gives an error if no object file for your program is present in the current directory.

- Compile Via Assembler tells the compiler to produce assembly language output, then invoke TASM to assemble the output. The output is not contained in an .ASM file. To generate an .ASM file when using this setting, also turn on the Generate Assembler Source setting.

- Standard Stack Frame generates standard function entry and exit code. This simplifies the process of tracing back through the stack of called subroutines while debugging.

If you compile a source file with Standard Stack Frame off, any function that does not use local variables and has no parameters is compiled with abbreviated entry and return code. This makes the code shorter and faster, but prevents the debugger from "seeing" the function in the call stack. Thus, you should always turn Standard Stack Frame on when you compile a source file for debugging.

This setting is automatically turned off when you turn optimizations on with the Smallest Code or Fastest Code buttons in the Compiler | Optimizations subsection. A check box for this setting is also located in the Compiler | Optimizations subsection. These two settings are always the same; that is, if the box in the Optimizations subsection is turned on, so is the one in the Code Generation subsection.

- Generate Underbars automatically adds an underbar, or underscore, character ( _) in front of every global identifier (such as function names and global variables). If you are linking with the standard Borland libraries, this setting must be turned on.

Debugging Options

The Debugging Options box contains settings that specify what type of debugging information the compiler should include in object files.

While including debugging information in your program doesn't affect execution speed, it does affect compilation time. Including debugging information requires longer compilation times. It also results in larger object files.

- Line Numbers Debug includes line numbers in the object and object map files for use by a symbolic debugger.

- Debug Info In OBJs controls whether debugging information is included in object (.OBJ) files. This setting is on by default. You need debugging...
information in object files in order to use either the integrated debugger or the standalone Turbo Debugger.

■ Test Stack Overflow generates code to check for a stack overflow at runtime. This costs space and time in a program, but helps prevent elusive stack overflow bugs.

Use the Defines input box to enter macro definitions for the preprocessor. Separate multiple defines with semicolons (;) and assign values with an equal sign (=); for example,

```
TESTCODE; PROGCONST=5
```

Leading and trailing spaces are stripped, but embedded spaces are left intact. To include a semicolon in a macro, place a backslash (\) in front of it.

If you press Alt+↓ when the cursor is in the Defines box, or click the arrow on the right side of the Defines box, a history list drops below the box. This list displays the last seven definitions you’ve entered. Choose a definition from the list by double-clicking it or selecting it with the arrow keys and pressing Enter.

The Calling Convention settings let you specify the default conventions used to pass arguments and call functions. You can specify using the C (\_cdecl), Pascal (\_pascal), Register (\_fastcall), or Standard (\_stdcall) calling sequence. The default is Standard. The differences between these calling conventions are in the way each handles stack cleanup, order of parameters, case, and prefix (underbar) of global identifiers.

For more information about the \_fastcall calling convention, see Appendix A, “The optimizer.” For information about all of the calling conventions, see Chapter 2, “Language structure,” in the Programmer’s Guide.

The C++ Options subsection contains settings that tell the Borland C++ compiler how to prepare object code when compiling C++ code, including how to handle inline functions and pointers, and how to generate C++ virtual tables and templates.

Borland C++ supports three different kinds of member pointer types. You can control what pointer types the compiler accepts with the C++ Member Pointers radio buttons:

■ Support All Cases places no restrictions on which members can be pointed to. Member pointers use the most general (but not always the most efficient) representation.
Compiler|C++ Options

- Support Multiple Inheritance lets member pointers point to members of multiple inheritance classes, with the exception of members of virtual base classes.
- Smallest For Class specifies that member pointers use the smallest possible representation that lets member pointers point to all members of their particular class.
- Support Single Inheritance lets member pointers point to members of base classes that use single inheritance only.

**Use C++ Compiler**

The Use C++ Compiler radio buttons tell Borland C++ whether to always compile your programs as C++ code, or to always compile your code as C code except when the file extension is .CPP.

**Out-of-line Inline Functions**

Turn on Out-of-line Inline Functions when you want to step through or set breakpoints inside inline functions declared with the `inline` keyword. When Out-of-line Inline Functions is turned off, functions are expanded `inline` as you would expect. But when this setting is turned on, `inline` functions are called just like normal functions.

**C++ Virtual Tables**

The C++ Virtual Tables radio buttons let you control C++ virtual tables and the expansion of inline functions when debugging.

- The Smart setting generates common C++ virtual tables and out-of-line `inline` functions across modules within your application. As a result, only one instance of a given virtual table or out-of-line `inline` function is included in the program. This produces the smallest and most efficient executables.
- The Local setting generates local virtual tables and out-of-line `inline` functions. As a result, each module gets its own private copy of each virtual table or out-of-line `inline` function it uses; this setting produces larger executables than the Smart setting.
- The External setting generates external references to virtual tables; one or more of the modules that make up the program must be compiled with public virtual tables to supply the definitions for the virtual tables.
- The Public setting generates public definitions for virtual tables.

**Template Generation**

The Template Generation settings let you specify how Borland C++ generates template instances in C++. For more information about templates, see Chapter 3, "C++ specifics," in the Programmer's Guide.

- Smart generates public (global) definitions for all template instances. If more than one module generates the same template instance, the linker
automatically merges duplicates to produce a single definition. This setting is on by default, and is normally the most convenient way of generating template instances. The Smart setting is equivalent to the \texttt{-Jg} command-line option.

- Global, like Smart, generates public definitions for all template instances. However, it does not merge duplicates, so if the same template instance is generated more than once, the linker reports public symbol redefinition errors. The Global setting is equivalent to the \texttt{-Jgd} command-line option.

- External generates external references to all template instances. If you use this setting, you must make certain that the instances are publicly defined elsewhere in your code. The External setting is equivalent to the \texttt{-Jgx} command-line option.

The Optimizations subsection lets you specify which optimizations (if any) you want performed on your program. For more information on optimization, see Appendix A, “The optimizer.”

Using the Optimizations box you can choose which specific optimizations you want enabled or disabled:

- Dead Storage Elimination eliminates stores into dead variables. This setting corresponds to the \texttt{-Ob} command-line option.

- Local Common Expressions enables common expression elimination within basic blocks only. This setting corresponds to the \texttt{-Oc} command-line option.

- Global Optimizations enables common subexpression elimination within an entire function. This setting corresponds to the \texttt{-Oz} command-line option.

- Global Register Allocation enables global register allocation and variable live-range analysis. This setting corresponds to the \texttt{-Oe} command-line option.

- Assume No Pointer Aliasing instructs the compiler to assume that pointer expressions are not aliased in common subexpression evaluation. This setting corresponds to the \texttt{-Oa} command-line option.

- Intrinsic Expansion enables inlining of intrinsic functions such as \texttt{memcpy}, \texttt{strlen}, and so on. The functions that can be inlined in this manner are listed on page 137. This setting corresponds to the \texttt{-Oi} command-line option.

- Standard Stack Frame is automatically turned off when you turn optimizations on with the Smallest Code or Fastest Code buttons. A check box for this setting is also located in the Compiler | Code...
Generation subsection. These two settings are always the same; that is, if the box in the Optimizations subsection is turned on, so is the one in the Code Generation subsection. This setting does not have a command-line equivalent.

**Optimize For**

The Optimize For buttons let you change Borland C++’s code-generation strategy. Normally the compiler optimizes for size, choosing the smallest code sequence possible. You can also have the compiler optimize for speed, so that it chooses the fastest sequence for a given task. For creating PM applications, you’ll probably want to optimize for speed.

**Minimal Opts**

The Minimal Opts button turns off as many optimizations as possible. In effect, all optimizations are turned off, and Standard Stack Frame is turned on.

**Smallest Code**

The Smallest Code button turns on a set of optimizations that is designed to produce the smallest possible code size. The optimizations that are turned on are dead storage elimination and local common expressions.

**Fastest Code**

The Fastest Code button turns on a set of optimizations that is designed to produce the fastest possible executable code. The optimizations that are turned on are dead storage elimination, local common expressions, global optimizations, global register allocation, and intrinsic expansion.

**Source Options**

The settings in the Source Options subsection tell the compiler to expect certain types of source code.

**Keywords**

The Keywords radio buttons tell the compiler how to recognize keywords in your programs:

- Choosing Borland C++ tells the compiler to recognize the Borland C++ extension keywords, including \_asm, \_cdecl, \_export, \_far16, \_pascal, \_fastcall, and the register pseudovariables (_AX, _BX, and so on). For a complete list, refer to Chapter 1, “Lexical elements,” in the *Programmer’s Guide*.

- Choosing ANSI tells the compiler to recognize only ANSI keywords and treat any Borland C++ extension keywords as normal identifiers.

- Choosing UNIX V tells the compiler to recognize only UNIX V keywords and treat any Borland C++ extension keywords as normal identifiers.
Choosing Kernighan And Ritchie tells the compiler to recognize only the K&R extension keywords and treat any Borland C++ extension keywords as normal identifiers.

Nested Comments

The Nested Comments setting lets you nest comments in Borland C++ source files. Standard C implementations do not permit nested comments, which are not portable.

Identifier Length

Use the Identifier Length input box to specify the number $n$ of significant characters in an identifier. All identifiers are treated as distinct only if their first $n$ characters are distinct. This includes variables, preprocessor macro names, and structure member names. $n$ can be from 1 to 249. Specifying $n$ to be 0 or 250 forces the compiler to allow identifiers of unlimited length. The default is 0.

Messages

The Messages subsection lets you customize the behavior of compiler error and warning messages in the IDE, including what constitutes a fatal number of warnings and errors and which warning messages are displayed.

Pages 2 through 9 in the Messages subsection in the Settings notebook are warning checklist pages. Each page contains a list of warning messages that you can turn on and off. If a warning is turned off, it is not displayed when the compiler encounters it.

To turn a warning on, check the box next to the warning description. To turn a warning off, uncheck the box next to the warning description.

Next to each warning is a three-letter code. You can use this code with the `-wxxx` command-line option to turn warnings on and off on the command line. For information on the `-wxxx` option, see page 117.

Display Warnings

The Display Warnings box lets you specify how you want error messages to be handled:

- The Display Warnings settings let you choose whether the compiler displays all warnings, only the warnings selected in the Messages submenu setting, or no warnings.
- Errors: Stop After causes compilation to stop after the specified number of errors has been detected. The default is 25, but you can enter any number from 0 to 255. Entering 0 causes compilation to continue until the end of the file or until the warning limit entered below has been reached, whichever comes first.
Warnings: Stop After causes compilation to stop after the specified number of warnings has been detected. The default is 100, but you can enter any number from 0 to 255. Entering 0 causes compilation to continue until the end of the file or until the error limit entered above has been reached, whichever comes first.

Portability

The Portability box on page 2 of the Messages subsection in the Settings notebook lets you specify which types of portability problems you want to be warned about.

ANSI Violations

The ANSI Violations boxes on pages 3 and 4 of the Messages subsection in the Settings notebook let you specify which violations of the ANSI specification you want to be warned about.

C++ Warnings

The C++ Warnings boxes on pages 5 and 6 of the Messages subsection in the Settings notebook let you specify which C++ warnings you want to be warned about.

General

The General boxes on pages 7, 8, and 9 of the Messages subsection in the Settings notebook let you specify which miscellaneous warnings you want to be warned about.

Names

The Names subsection lets you change the default segment, group, and class names for code, data, BSS, and far data sections. You can also specify a name for the virtual table segment and class for C++ programs. Do not change the settings in this subsection unless you are an expert and have read Chapter 11, "OS/2 memory management," in the Programmer's Guide.

Segment Names

To name a segment something besides the default name, click in the box corresponding to the segment you want to name and type in the desired name.

If you press Alt+↓ when the cursor is blinking in a name box, a history list drops down below the box. This list displays the last seven names you’ve entered. To choose a name from the list, double-click it with the mouse, or select it with the arrow keys and press Enter.

Make section

The Make section lets you specify how the IDE makes your current project or module.
Use the Break Make On radio buttons to set the condition that stops the making of a project. The default is to stop after compiling a file with errors. However, you can make the IDE stop making a project after warnings by turning on the Warnings setting. You can also force the IDE to continue the make process (as long as it encounters no fatal errors) by turning on the Force Errors setting. Lastly, you can stop the make process after compiling all sources, but before linking by turning on the All Sources Processed setting.

When the Check Auto-dependencies setting is checked, the Project Manager automatically checks dependencies for every .OBJ file on disk that has a corresponding .CPP or .C source file in the project list.

The Project Manager opens the .OBJ file and looks for information about files included in the source code. This information is always placed in the .OBJ file by the Borland C++ IDE, as well as the command-line version of Borland C++ when the source module is compiled. Then every file that was used to build the .OBJ file is checked for time and date against the time and date information in the .OBJ file. The source file is recompiled if the dates are different. This is called an autodependency check. If this setting is turned off, no file checking is performed.

After a source file is successfully compiled, the project file contains valid dependency information for that file. Once that information is in the project file, the Project Manager uses it to do its autodependency check. This is much faster than reading each .OBJ file.

This section of the Settings notebook lets you specify the type of application you want to produce, how you want to handle import and export functions, and whether you want to link to single- or multi-thread libraries.

The Program Target box has five buttons:

- PM Exe specifies that the program should be compiled and linked for execution in the PM environment.
- OS2 Exe specifies that the program should be compiled and linked as an OS/2 application. You can run this type of application in a full-screen OS/2 shell or in a windowed OS/2 shell under PM.
- OS2 DLL specifies that the program should be compiled and linked as an OS/2 DLL.

- OS2 Lib specifies that the file should be compiled and linked as an OS/2 library file.

- Text Mode App specifies that the file should be compiled and linked as a full-screen text mode application. Programs compiled with this setting cannot be run in a windowed OS/2 shell under PM.

**Thread Options**

The Thread Options buttons cause the compiler to generate code for either a single-thread executable or a multi-thread executable. This also dictates which libraries your program is linked with. For a discussion of multi-thread programming, see Chapter 9, “Building OS/2 applications,” in the Programmer’s Guide.

**Generate Import Library**

The Generate Import Library buttons control when and how IMPLIB is executed during the MAKE process. The Use DLL setting generates an import library that consists of the exports in the DLL. The Use DEF setting generates an import library of exports in the DEF file. If either of these settings is checked, MAKE invokes IMPLIB after the linker has created the DLL. This setting controls how the transfer macro $IMPLIB gets expanded.

**Linker section**

The Linker section lets you configure various settings to be passed to the linker, such as whether to include debugging information, case sensitivity, and which libraries to link with. The Linker section contains two subsections, Link Settings and Link Libraries.

**Link Settings**

The Link Settings subsection (labeled Linker on the subsection tab) contains a number of settings that specify how Borland C++ should link your application.

**Options**

The Options box contains a number of settings specifying how your program should be linked:

- Include Debug Info controls whether debugging information is linked into the .EXE file. Turn this setting on to use the symbolic debugging capabilities of either the integrated debugger or the standalone Turbo Debugger.

  Turning this setting off results in smaller, more compact executable files. Once you have finished debugging your program you should relink it.
without debugging information so that your final executable is as small as possible.

- The Image Is Based setting specifies whether an application has an image base address. If this setting is turned on, internal fixups are removed from the image and the requested load address of the first object in the application is set to the number specified in the Base Address input box. This can greatly reduce the size of your final application module. It is not recommended for use when producing a DLL.

- Case-Sensitive Link affects whether the linker is case-sensitive. Normally, this setting should be checked, because C and C++ are both case-sensitive languages.

- Case-Sensitive Exports affects whether the linker is case-sensitive in regard to the names in the IMPORTS and EXPORTS sections of the module definition file. By default, the linker ignores the case of these names. This setting is probably useful only when you are trying to export non-callback functions from DLLs—such as exported C++ member functions.

- The Base Address setting specifies the load address to be used for your application. This address is disregarded unless the Image Is Based setting is turned on. Note that the address is specified in multiples of 64K (0x10000). Thus, to load an image at 64K you would set the Base Address setting to 1.

  Because OS/2 loads all .EXE images at 64K, we advise you to link all .EXEs with the Base Address setting set to 1.

- The File Alignment setting specifies page alignment for code and data within the executable file. The value is interpreted as a decimal power of 2. For example, if you set the File Alignment to 12, the pages of code and data will be stored on 4096-byte boundaries. OS/2 seeks pages for loading based on this alignment value.

Use the Map File radio buttons to choose the type of map file Borland C++ produces:

- Off instructs Borland C++ not to produce a map file. This is equivalent to the TLINK /x option.

- Segments instructs Borland C++ to list only the segments in the program, the program start address, and any warning or error messages produced during the link. This is equivalent to the default map produced by TLINK.

- Publics instructs Borland C++ to create a map file with program segments, the program start address, error and warning messages, and add a list of public symbols. This is equivalent to the TLINK /m option.
Detailed instructs Borland C++ to create a map file like that created with the Publics button, but with the addition of a detailed segment map. This is equivalent to the TLINK /s option.

For settings other than Off, the map file is placed in the output directory defined in the Directories section.

The Link Libraries subsection contains a number of settings that specify which libraries Borland C++ should link with your application.

The Borland C++ standard run-time libraries are available in both Static (.LIB) and Dynamic (.DLL) form. Choosing the dynamic form can help reduce the size of your executable file, and can also reduce the overhead of loading libraries more than once if they are called by more than one application running simultaneously.

Choosing None forces the linker to link only the file you are currently working on, or files listed in the project file if you are working on a project. If you choose None, you must provide all the functions required by the program, including entry/exit code.

The container class libraries are available in both Static (.LIB) and Dynamic (.DLL) form. Choosing the dynamic form can help to reduce the size of your executable file, and can also reduce the overhead of loading these libraries more than once if they are called by more than one application running simultaneously.

Choose None if you are not using the container class libraries. If you choose Static or Dynamic, and you are not using container class libraries, there's no harm caused, but linking is appreciably slower.

The Link Warnings subsection lets you specify which linker error and warning messages are displayed in the IDE.

The Librarian section lets you make several settings affecting the use of the Librarian. The Librarian combines the .OBJ files in your project into a .LIB file.
The Options box contains a number of settings that tell the librarian program how to build your library:

- **Generate List File** determines whether the Librarian automatically produces a list file (.LST) listing the contents of your library when it is created.
- **Case-Sensitive Library** tells the Librarian to treat case as significant in all symbols in the library (this means that CASE, Case, and case, for example, would all be treated as different symbols).
- **Purge Comment Records** tells the Librarian to remove all comment records from modules added to the library.
- **Create Extended Dictionary** determines whether the Librarian includes, in compact form, additional information that helps the linker process library files faster.

The Library Page Size setting lets you set the number of bytes in each library "page" (dictionary entry). The page size determines the maximum size of the library: it cannot exceed 65,536 pages. The default page size, 16, permits a library of about 1 MB in size. To create a larger library, change the page size to the next higher value (32). The page size must be a power of 2 ($2 = 2^1, 4 = 2^2, 8 = 2^3$, and so on).

**Debugger Options section**

The Debugger Options section contains settings you can use to configure the behavior of the IDE integrated debugger.

The Debugger Options subsection contains settings that let you specify how the Borland C++ integrated debugger performs in the IDE environment.

The PM Debugging Mode settings let you set which mode the integrated debugger operates in. Hard Mode sets the debugger to operate in hard mode; Soft Mode sets the debugger to operate in soft mode.

The difference between hard mode and soft mode is that in hard mode the debugger traps all PM messages. In effect, this turns PM into a single-tasking system controlled by the debugger. In soft mode, other processes receive messages normally, with the debugger intercepting only its own messages and messages to the process being debugged.
The Use Evaluator setting affects how expressions are evaluated in IDE debugger input boxes. This lets you enter expressions using the most convenient syntax. This setting does not affect the source language of your code, and does not need to be the same language as your source code. You can control what syntax is used with the Use Evaluator radio buttons:

- C Evaluator
- C++ Evaluator
- TASM Evaluator

The Popup On Exception box lets you specify which pop-up windows (known as views) you want opened when the IDE encounters an exception in your code. These views are identical to those used in Borland's Turbo Debugger. There are four views that you can specify:

- Call Stack View displays a view of the program call stack.
- Source View displays a view of the program source code with the line where the exception occurred highlighted. This view is separate from source display in the IDE editor.
- Disassembly View displays a view of the disassembled program code with the instruction where the exception occurred highlighted.
- Local Variable View displays a view of all variables within the scope of the current function along with their values at the time the exception occurred. Variables with a greater scope (such as class members and global variables) are not displayed.

The Action On Messages radio buttons let you specify the action the IDE should take when presenting a message. There are four action settings:

- The Use Smart Messages button specifies that the IDE use smart messages. With smart messages, the IDE opens a window for some messages and beeps for others. The message is also displayed on the IDE status line.
- The Beep And Show Message setting tells the IDE to beep and display the message in a pop-up window.
- The Show Message setting tells the IDE to display the message in a pop-up window.
- The Beep setting tells the IDE to beep only. The message is not displayed.
The Disassembly View Local Options subsection contains settings that affect how the disassembly view behaves.

The Disassembly View settings box contains settings that affect how your disassembled code is displayed.

- When the Follow PC setting is checked, the debugger updates the open views as necessary to show the source and disassembly code associated with the program counter (the "PC") as you step or run through your application.
- The Show Symbolic setting, when checked, shows identifiers (symbols) as addresses in the disassembly pane. For example, when shown as symbolic disassembly, the instruction `CALL WndProc`, where `WndProc` is a valid symbolic name, appears as `CALL 00010663`.
- When the Show Source setting is checked, the debugger displays the line of source code associated with each set of disassembled instructions.

The Include Views settings box lets you specify which panes you want to appear in the disassembly view:

- The Memory setting displays the memory pane in the disassembly view. The memory pane shows a raw display of an area of memory. Each line in this pane displays the following information:
  - On the left: The address of the data.
  - In the middle: The raw display of one or more data items. The format of this area depends on the display format you choose with Display Memory As settings.
  - On the right: The display characters that correspond to the data bytes displayed, unless you choose Display Memory As Stack setting. The debugger displays all printable byte values as their display equivalents; any nonprintable characters appear as dots (.). When you first open this view, the memory pane displays memory as byte values. The ASCII representation of the bytes appears to the right of the byte values.
- The Stack setting displays the Stack pane in the disassembly view.
- The Registers setting displays the registers pane in the disassembly view. The registers pane displays the contents of each of the 16 CPU registers. To edit the value of a CPU register's contents, double-click the register (or select the register and press Enter). The Change Register NNN (where NNN is the name of the register you're modifying) dialog box opens up.
The Flags setting displays the flags pane in the disassembly view. The flags pane shows the state of the eight CPU flags. Each flag is indicated by a single letter:
- C (Carry)
- P (Parity)
- A (BCD carry)
- Z (Zero)
- S (Sign)
- I (Interrupt)
- D (Direction)
- O (Overflow)

The flags show the result of the last logical or arithmetic operation that the CPU performed. To change the state of a CPU flag, double-click the flag (or select the flag and press Enter).

The Display Memory As Bytes settings box lets you specify how you want the data in the memory pane of the disassembly view displayed. There are a number of formats you can use:
- Byte
- Double
- Short
- Stack
- Long

The Variables View Local Options subsection contains a number of settings that let you configure how information appears in the variable view.

You can configure the Variable view to display a list of one of three types of variable:
- Global variables
- Local variables
- Function entries

The Variable Information buttons let you modify the type of information displayed in the Variable view:
- The Stack button opens a second pane in the Variable view that displays the current call stack.
The Argument Names In Stack button displays the names of the arguments to each function listed in the call stack pane. This has an effect only if the Stack button is set on.

The Argument Values In Stack button displays the value of the arguments to each function listed in the call stack pane. This has an effect only if the Stack button is set on.

The Show Address And Type button displays the memory address and variable type of each variable displayed in the Variable view.

The Display Selected Item As list box lets you specify the format in which you want the selected variable.

The Call Stack View Local Options subsection contains a number of settings that let you configure how information appears in the call stack view.

The Call Stack Will Show buttons let you configure the display in the Call Stack view.

The Call Stack Will Show buttons let you configure the display in the Call Stack view.

The Type Information button displays the value and variable type of each variable displayed in the watch view.

The Stack button opens a second pane in the watch view that displays the current call stack.
The Argument Names In Stack button displays the names of the arguments to each function listed in the call stack pane. This has an effect only if the Stack button is set on.

The Argument Values In Stack button displays the value of the arguments to each function listed in the call stack pane. This has an effect only if the Stack button is set on.

The Format Of Selected Item list box lets you specify the format in which you want the selected variable.

The Evaluator View Local Options subsection contains a number of settings that let you configure how information appears in the evaluator view.

The Evaluator Show settings box contains check boxes that you can use to configure what information appears in the evaluator view.

- The Type Information button displays the value and variable type of each variable displayed in the evaluator view.
- The Stack button opens a second pane in the evaluator view that displays the current call stack.
- The Argument Names In Stack button displays the names of the arguments to each function listed in the call stack pane. This has an effect only if the Stack button is set on.
- The Argument Values In Stack button displays the value of the arguments to each function listed in the call stack pane. This has an effect only if the Stack button is set on.

The Format Of Selected Item list box lets you specify the format in which you want the selected variable.

The Inspector View Local Options subsection contains settings that let you configure how information appears in the inspector view.

The Show Type Information button displays the value and variable type of each variable displayed in the inspector view.

The Format Of Selected Item list box lets you specify the format in which you want the selected variable.

The Memory View Local Options subsection contains settings that let you configure how information appears in the memory view.
The Memory Will Show settings box contains check boxes that you can use to configure what information appears in the memory view.

- The Stack button opens a second pane in the memory view that displays the current call stack.
- The Argument Names In Stack button displays the names of the arguments to each function listed in the call stack pane. This has an effect only if the Stack button is set on.
- The Argument Values In Stack button displays the value of the arguments to each function listed in the call stack pane. This has an effect only if the Stack button is set on.
- The Frame Registers button displays the location and frame address for each stack frame.

When the Memory View Follows Stack setting is set on, the contents of the memory view are updated to reflect the current position of the stack pointer and the contents of the stack.

The Display Memory As Bytes settings box lets you specify how you want the data in the memory view displayed. There are a number of formats you can use:

- Byte
- Double
- Float
- Long
- Long Double
- Short
- Stack

The Register View Local Options subsection contains settings that let you configure how information appears in the register view.

You can display two different panes in the register view: the register pane, which displays the contents of the CPU registers, and the flags pane, which displays the contents of the CPU flags register. You can set each of these panes on or off with the Registers and Flags settings.

You can have the register view display the registers and flags panes in one of two formats, Horizontal or Vertical, by pressing the appropriate button. When the Register Layout is Horizontal, the flags pane is placed to the right of the register pane. When the Register Layout is Vertical, the flags pane is placed beneath the register pane. This setting only has an effect when both the Register and Flags settings are set on.
### Debugger Options | Register View Local Options

You can display the contents of the registers in one of three formats: Decimal, Hexadecimal, or Octal, by pressing the appropriate button.

The File And Numeric View Local Options subsection contains settings that let you configure how information appears in the file and numeric processor views.

The File View Will Display As settings box lets you specify how you want the data in the file view displayed. There are a number of formats you can use:

- Byte
- Double
- Float
- Long
- Long Double
- Short
- Stack

Setting the Display ASCII In File View setting on tells the file view to display the corresponding ASCII character along with the data displayed in the numeric format specified by the File View Will Display As setting.

You display the contents of the numeric processor view in one of two formats, Decimal or Hexadecimal, by pressing the appropriate button.

### Directories section

The Directories section tells Borland C++ where to find files, including the files it needs to compile, link, and debug, and where to put output files produced by the Borland C++ tools. This section contains four input boxes:

- **Include box**: Specifies which directories contain include files. Standard include files are those given in angle brackets (<> in an \#include statement (for example, \#include <myfile.h>). The Include directories are also searched for files in quoted \#include statements (such as \#include "myfile.h") which are not found in the current directory. Multiple directory names are permitted, separated by semicolons.

- **Library box**: Specifies the directories that contain your Borland C++ start-up object files (.OBJ files) and run-time library files (.LIB files) and any other libraries your project uses. Multiple directory names are permitted, separated by semicolons.
■ The Output box specifies the directory that stores your .OBJ, .EXE, and .MAP files. Borland C++ looks for and writes files to that directory when doing a make or run, and checks dates and times of .OBJs and .EXEs. If the entry is blank, the files are stored in the current directory. Multiple directory names are not permitted.

■ The Debug Source box specifies the directories where the Borland C++ integrated debugger looks for the source code for modules that do not belong to the open project (for example, container class libraries). Multiple directories can be entered, separated by semicolons. If the entry is blank, the current directory is searched.

Use the following guidelines when entering directories in these input boxes:

■ Separate multiple directory path names with a semicolon (;).
■ You can use up to a maximum of 256 characters (including whitespace).
■ You can place whitespace before and after the semicolon, but this is not required.
■ Relative and absolute path names are permitted, including path names relative to the logged position in drives other than the current one. Here’s an example:

\[C:\;C:\LIB;C:\MYLIBS;A:\BORLANDC\MATHLIBS;A:\..\VIOLIBS\]

Environment section

The Environment section lets you tailor the Borland C++ IDE to perform the way you want it to perform. This section contains four subsections that let you modify the characteristics, or “look and feel,” of the editor, desktop, and IDE traits.

Preferences

The Preferences subsection contains settings that let you specify the general behavior of the IDE, such as autosave settings, editor key bindings, and some others.

Editor Key Bindings

The Editor Key Bindings list box lets you change the configuration of the keyboard shortcuts in the IDE editor by compiling a Turbo Editor Macro Language (TEML) file using the Turbo Editor Macro Compiler (TEMC). TEMC and TEML files are fully described in the online document UTIL.DOC.

When the cursor is in the list box you can press letters to go to the next TEML file that begins with that letter. If you press Alt↓ when the cursor is
in the Editor Key Bindings box, or click the arrow on the right side of the box, a list drops below the box. This list displays the available TEML files. Choose a file name from the list by double-clicking it or by selecting it with the arrow keys and pressing Enter.

Only files with the extension .TEC that are located in the BIN directory of your Borland C++ installation are displayed in the Editor Key Bindings list box.

If you're used to the command set from previous versions of Borland C++ or Turbo C++, you can load ALT.TEC. This file provides editor key bindings that are compatible with the Alternate command set used in many other Borland products.

The AutoSave box lets you specify which parts of your current project you want the IDE to automatically save. Each part is saved under different conditions, but is always saved (if its setting is set on in the AutoSave box) when you exit Borland C++.

- The Environment setting specifies that all the settings you made in this session are saved automatically when you exit Borland C++.
- The Editor Files setting specifies that any source file in your current project that has been modified since the last time you saved it is saved whenever you run your program.
- The Desktop setting specifies that the desktop configuration is saved when you close a project or exit Borland C++. The desktop is restored when you reopen the project or return to Borland C++.
- The Breakpoints setting specifies that the current breakpoint settings are saved when you close a project or exit the IDE. These breakpoints are restored when you reopen the project or return to Borland C++.
- The Project setting specifies that all your project, autodependency, and module settings are saved when you close your project or exit. They are restored when you reopen the project or return to Borland C++.

The Code Page setting lets you specify which video code page you want to use. The default is 437, the standard U.S. code page.

The Source File radio buttons let you specify which copy of your current source files you want to use for compilation if the source in the edit buffer has changed since the current project build.

- Use Source File On Disk specifies that Borland C++ should use the file saved to disk instead of the file contained in the edit buffer.
Use Source File In Buffer specifies that Borland C++ should use the file contained in the edit buffer instead of the file saved to disk.

Prompt For File To Use specifies that Borland C++ should ask you which version of the file you want to use.

Source Tracking

The New Window setting specifies that the IDE opens a new window if it encounters a source file that is not open in an editor window while stepping through source or viewing the source from the Transcript window. Selecting Current Window causes the IDE to replace the contents of the active Edit window with the new file instead of opening a new Edit window.

Save Old Messages

When Save Old Messages is set on, Borland C++ saves the error messages currently in the Transcript window, appending any messages from further compiles to the window. Messages are not saved from one session to the next. By default, Borland C++ automatically clears messages before a compile, a make, or a transfer that uses the Transcript window.

Desktop

The desktop includes the configuration of open files, icons, windows, and their arrangement in the Borland C++ IDE desktop. The Desktop subsection lets you specify what portions of the desktop you want saved and where you want to position the SpeedBar.

Save

The Save box lets you set whether history lists, the contents of the Clipboard, and the locations and contents of open and closed windows are saved across sessions.

SpeedBar

The buttons in the SpeedBar box let you configure where you want the SpeedBar to be located, or if you even want it active at all. Pressing Off means that the SpeedBar is not present in the IDE at all. Pressing Popup makes the SpeedBar a floating palette. Pressing Horizontal Bar places the SpeedBar at the top of the desktop window, running from left to right. Pressing Vertical Bar places the SpeedBar on the left side of the desktop window, running from top to bottom.

Editor

The Editor subsection contains settings that let you tailor the behavior of the IDE editor.

Editor Options

The Editor Options box lets you customize the following editor attributes:
When Create Backup Files is checked (the default), Borland C++ automatically creates a backup of the source file in the Edit window when you choose File I Save and gives the backup file the extension .BAK.

When Syntax Highlighting is checked, you can control the colors in an Edit window of various elements in your C or C++ code. You can set these colors in the Environment | Syntax Hilite subsection.

When Autoindent Mode is turned on, pressing Enter in an Edit window positions the cursor under the first nonblank character in the preceding nonblank line. This can be a great aid in typing readable program code.

When Use Tab Character is checked, Borland C++ inserts a true tab character (ASCII 9) when you press Tab. When this setting is not checked, Borland C++ uses spaces instead. The size of a tab character is determined by the Tab Size setting.

When you turn on Optimal Fill, Borland C++ begins every autoindented line with the minimum number of characters possible, using tabs and spaces as necessary. This produces lines with fewer characters than when Optimal Fill is set off.

When Backspace Unindents is set on (which is the default) and the cursor is on a blank line or the first non-blank character of a line, the Backspace key aligns (outdents) the line to the previous indentation level. This setting is effective only when Cursor Through Tabs is also set on.

When you set Cursor Through Tabs on, the arrow keys move the cursor by space through tabs; otherwise the cursor jumps over tabs.

When Group Undo is set off, choosing Edit | Undo reverses the effect of a single editor command or keystroke. For example, if you type ABC, it takes three Undo commands to delete C, then B, then A.

If Group Undo is checked, Undo reverses the effects of the previous command and all immediately preceding commands of the same type. The types of commands that are grouped are insertions, deletions, overwrites, and cursor movements. For example, if you type ABC, one Undo command deletes ABC.

For the purpose of grouping, inserting a carriage return is considered part of the same command as typing text. For example, if you press Enter, then type ABC, choosing Undo deletes ABC and the carriage return and moves the cursor back to the original line. (See page 41 for more information about Undo.)

When Persistent Blocks is set on, marked blocks behave as they always have in Borland’s C and C++ products; that is, they remain marked until deleted or unmarked (or until another block is marked). With this setting off, moving the cursor after a block is selected deselects the entire block of text.
When Overwrite Blocks is set on and Persistent Blocks is set off, marked blocks behave differently in these instances:

1. Pressing the Del key or the Backspace key clears the entire selected text.
2. Inserting text (pressing a character or pasting from the Clipboard) replaces the entire selected text with the inserted text.

---

**Tab Size**

If you check Use Tab Character in the Editor Options box and press Tab, Borland C++ inserts a tab character in the file and the cursor moves to the next tab stop. The Tab Size input box lets you dictate how many characters to move for each tab stop. Legal values are 2 through 16; the default is 8.

To change the way tabs are displayed in a file, just change the Tab Size value to the size you prefer. Borland C++ redisplays all tabs in that file in the size you chose. You can save this new tab size in your configuration file by selecting Save on the Project menu. When the Save Options dialog box is open, make sure the Project box is checked, and press OK.

---

**Default Extension**

The Default Extension input box lets you tell Borland C++ which extension to use as the default when compiling and loading your source code. Changing this extension doesn't affect the history lists in the current desktop.

---

**Fonts**

The Fonts subsection lets you change the attributes and style of the font used in the IDE's edit windows. These font changes do not affect the other IDE windows.

---

**Name**

The Name box lets you enter the name of the font you want to use in your edit windows.

If you press Alt+↓ when the cursor is in the Name box, or click the arrow on the right side of the Name box, a choice list drops below the box. This list displays all the fonts that are available to the IDE. Choose a definition from the list by double-clicking it or selecting it with the arrow keys and pressing Enter.

---

**Size**

The Size box lets you enter the size of the font you want to use in your edit windows.

If you press Alt+↓ when the cursor is in the Size box, or click the arrow on the right side of the Size box, a choice list drops below the box. This list displays all the sizes you can use with the current font. Choose a definition
from the list by double-clicking it or selecting it with the arrow keys and pressing Enter.

The Style buttons let you specify what style the chosen font appears in an edit window. You can choose either Bold, Italic, both, or neither.

The Syntax Hilite subsection contains controls that let you customize the colors the editor uses to represent language elements in your code.

The Element list box lists the elements of C and C++ code that are represented by different colors. You can use the Element box to select the current syntax element, and then customize the color and attributes used to display that element.

When the cursor is in the list box you can press letters to go to the next language element that begins with that letter. If you press Alt+↓ when the cursor is in the Element box, or click the arrow on the right side of the box, a list drops below the box. This list displays the language elements for which you can select colors. Choose a definition from the list by double-clicking it or selecting it with the arrow keys and pressing Enter.

The Elements list box contains the following syntax elements:

- Breakpoint
- Character
- Comment
- CPU position
- Float
- Hex
- Identifier
- Illegal char
- Integer
- Octal
- Preprocessor
- Reserved word
- String
- Symbol
- Whitespace

When the Default FG setting is on, the IDE displays the current syntax element in the default foreground color (usually black), regardless of the color chosen in the FG box.

When the Default BG setting is on, the IDE displays the current syntax element in the default background color (usually white), regardless of the color chosen in the BG box.
Use the **FG** box to select a foreground color for the current syntax element. Simply click on the desired color.

Use the **BG** box to select a foreground color for the current syntax element. Simply click on the desired color.

Click the **Bold** check box to make the current syntax element appear bold.

Click the **Italic** check box to make the current syntax element appear italicized.

Click the **Underline** check box to make the current syntax element appear underlined.

In the middle of the Syntax Hilite page there is a small code sample that contains an example of each different code element, and displays with the current colors and attributes assigned to that element. You can change the current syntax element by clicking it. The name in the Element box also changes to reflect the new current syntax element. Note that the code example does not contain all the code elements that are contained in the Element list box.

The extension box lets you specify which file types syntax highlighting is applied to. You can use standard OS/2 wildcards to specify file types. You can also specify more than one file type by stringing together a number of file specifications separated by semicolons. For example, if you want to use highlighting only on C and C++ files, your extension specification would look like this:

```
*.C;*.CPP
```

**Transfer section**

The Transfer section lets you customize which programs show up in the Transfer Items section of the Tools menu. You can also configure how parameters are passed to the programs in the Transfer Items list.
The Program Titles box lists all the transfer programs available on the Tools menu. If a title contains a tilde (~), the letter immediately after the tilde is used as the menu shortcut.

The Edit button opens the Modify/New Transfer Item dialog box. This dialog box lets you edit the characteristics of a transfer item. If an existing transfer item is highlighted when you select Edit, the input boxes in the Modify/New dialog box are automatically filled in; otherwise they’re blank.

Using the Modify/New dialog box, take these steps to add a new file to the Transfer dialog box:

1. Type a short description of the program you’re adding on the Program Title input box.
   Note that if you want your program to have a keyboard shortcut (like the S in the Save choice on the File menu or the t in the Cut choice on the Edit menu), you should include a tilde (~) in the name. Whatever character follows the tilde appears underlined on the Tools menu, indicating that you can press that key to choose the program from the menu.

2. Tab to Program Path and enter the program name and optionally include the full path to the program. (If you don’t enter an explicit path, only programs in the current directory or programs in your regular OS/2 path are found.)

3. Tab to Command Line and type any parameters or macro commands you want passed to the program. Macro commands always start with a dollar sign ($) and are entered in uppercase. For example, if you enter $CAP EDIT, all output from the program is redirected to a special Edit window in Borland C++.

4. If you want to assign a hot key, tab to the Hot Key settings and assign a shortcut to this program. Transfer shortcuts must be Shift plus a function key. Keystrokes already assigned appear in the list but are unavailable.

5. Now click or choose the New button to add this program to the list.

To modify an existing transfer program, highlight it in the Program Titles list of the Transfer dialog box, then choose Edit. After making the changes in the Modify/New Transfer dialog box, choose the Modify button.

The Translator check box in the Modify/New Transfer Item dialog box lets you put the Transfer program into the Project File Translators list (the list
you see when you choose Project \ Local Options). Check this setting when you add a transfer program that is used to build part of your project.

The Delete button removes the currently selected program from the list and the Tools menu.

Be careful with the Delete button! When you delete a transfer item, you cannot undo the delete!
Managing multi-file projects

Because most programs consist of more than one file, having a way to automatically identify those that need to be recompiled and linked would be ideal. Borland C++'s built-in Project Manager does just that and more.

The Project Manager lets you to specify the files belonging to the project. Whenever you rebuild your project, the Project Manager automatically updates the information kept in the project file. This project file includes

- All the files in the project.
- Where to find the files on the disk.
- The header files for each source module.
- Which compilers and command-line options need to be used when creating each part of the program.
- Where to put the resulting program.
- Code size, data size, and number of lines from the last compile.

Using the Project Manager is easy. To build a project,

1. Pick a name for the project file (from Project | Open Project).
2. Add source files using the Project | Add Item dialog box.
3. Tell Borland C++ to Compile | Make or press the Make button on the SpeedBar.

Then, with the project-management commands available on the Project menu, you can

- add or delete files from your project
- set options for a file in the project
- view included files for a specific file in the project

Let's look at an example of how the Project Manager works.
Sampling the Project Manager

Suppose you have a program that consists of a main source file, MYMAIN.CPP, a support file, MYFUNCS.CPP, that contains functions and data referenced from the main file, and myfuncs.h. MYMAIN.CPP looks like this:

```
#include <iostream.h>
#include "myfuncs.h"

int main(int argc, char *argv[])
{
  char *s;
  if(argc > 1)
    s=argv[1];
  else
    s="the universe";
  cout « GetString() « s « "\n";
  return 0;
}
```

MYFUNCS.CPP looks like this:

```
char ss[] = "The restaurant at the end of ">
char *GetString(void)
{
  return ss;
}
```

And myfuncs.h looks like this:

```
extern char *GetString(void);
```

These files make up the program that we'll now describe to the Project Manager.

The first step is to tell Borland C++ the name of the project file that you're going to use: Call it MYPROG.PRJ. Notice that the name of the project file is not the same as the name of the main file (MYMAIN.CPP). And in this case, the executable file will be MYPROG.EXE (and if you choose to generate it, the map file will be MYPROG.MAP).

Go to the Project menu and choose Open Project. This brings up the Open Project File dialog box, which contains a list of all the files in the current directory with the extension .PRJ. Because you're starting a new file, type in the name MYPROG in the Open Project File input box.

Notice that once a project is opened, the Add Item, Delete Item, Local Options, and Include Files options are enabled on the Project menu.
If the project file you load is in not in the current directory, the current directory is set to the directory that contains the project file.

You can keep your project file in any directory; to put it somewhere other than the current directory, just specify the path as part of the file name. (You must also specify the path for source files if they're in different directories.) Note that all files and corresponding paths are relative to the directory where the project file is loaded from. After you enter the project file name, you'll see a Project window.

The Project window contains the current project file name (MYPROG). Once you indicate which files make up your project, you'll see the name of each file and its path. When the project file is compiled, the Project window also shows the number of lines in the file and the amount of code and data in bytes generated by the compiler.

The SpeedBar shows which actions can be performed at this point: you can get help, add a file to the project, delete a file from the project, view include files required by a file in the Project, open an editor window for the currently selected file, compile a selected file in the project or build the entire project. Press the Add Item to Project button now to add a file to the project list.

The Add to Project List dialog box appears; this dialog box lets you select and add source files to your project. The Files list box shows all files with the .CPP extension in the current directory. (MYMAIN.CPP and MYFUNCS.CPP both appear in this list.) Three action buttons are available: Add, Done, and Help.

Because the Add button is the default, you can place a file in the Project window by typing its name in the Name input box and pressing Enter or by choosing it in the Files list box and choosing OK. You can also search for a file in the Files list box by typing the first few letters of the one you want. In this case, typing my should take you right to MYFUNCS.CPP. Press Enter. You'll see that MYFUNCS gets added to the Project window and then you're returned to the Add Item dialog box to add another file. Go ahead and add MYMAIN.CPP. Borland C++ will compile files in the exact order they appear in the project.

Close the dialog box and return to the Project window. Notice that the Lines, Code, and Data fields in the Project window show n/a. This means the information is not available until the modules are actually compiled.

After all compiler options and directories have been set, Borland C++ knows everything it needs about how to build the program called MYPROG.EXE using the source code in MYMAIN.CPP, MYFUNCS.CPP, and myfuncs.h. Now you'll actually build the project.
Choose Compile | Make or press the Make button on the SpeedBar to make your project. The output from the compile and link processes, such as error and warning messages, is displayed in the Transcript window. You can also open the Transcript window by choosing Tools | View Transcript.

Choose Run | Run to run your application. When you are done viewing the program output, close the application window.

When you leave the IDE, the project file you’ve been working on is automatically saved on disk; you can disable this by unchecking Project in the Environment | Preferences subsection of the Settings notebook.

The saved project consists of two files: the project file (.PRJ) and the desktop file (.DSK). The project file contains the information required to build the project’s related executable. The build information consists of compiler options, INCLUDE/LIB/OUTPUT paths, linker options, make options, and transfer items. The desktop file contains the state of all windows at the last time you were using the project.

The next time you use Borland C++, you can go right into your project by reloading the project file. Borland C++ automatically loads a project file if it is the only .PRJ file in the current directory; otherwise the default project and desktop (TCDEF.*) are loaded. Because your program files and their corresponding paths are relative to the project file’s directory, you can work on any project by moving to the project file’s directory and bringing up Borland C++. The IDE loads the correct files automatically. If no project file is found in the current directory, the default project file is loaded.

**Error tracking**

Syntax errors that generate compiler warning and error messages in programs can be selected and viewed from the Transcript window.

To see this, let’s introduce some syntax errors into the two files, MYMAIN.CPP and MYFUNCS.CPP. From MYMAIN.CPP, remove the first angle bracket in the first line and remove the c in char from the fifth line. These changes will generate five errors and two warnings in MYMAIN.

In MYFUNCS.CPP, remove the first r from return in the fifth line. This change will produce two errors and one warning.

Because you want to see the effect of tracking in multiple files, you need to modify the criterion Borland C++ uses to decide when to stop the make process. This is done by setting a radio button in the Make section of the Settings notebook.
Stopping a make

You can choose the type of message you want the make to stop on by setting one of the Break Make On options in the Make section of the Settings notebook. The default is Errors, which is normally the setting you'd want to use. However, you can have a make stop after compiling a file with warnings, with errors, or with fatal errors, or have it stop after all out-of-date source modules have been compiled.

The usefulness of each of these modes is really determined by the way you like to fix errors and warnings. If you like to fix errors and warnings as soon as you see them, you should set Break Make On to Warnings or maybe to Errors. If you prefer to get an entire list of errors in all the source files before fixing them, you should set the radio button to Fatal Errors or to Link. To demonstrate errors in multiple files, choose Fatal Errors in the Make section of the Settings notebook.

Syntax errors in multiple source files

Because you've already introduced syntax errors into MYMAIN.CPP and MYFUNCS.CPP, go ahead and choose Compile | Make to “make the project.” The Transcript window shows the files being compiled and the number of errors and warnings in each file and the total for the make. Choose OK when compiling stops.

Your cursor is now positioned on the first error or warning in the Transcript window. If the file that the message refers to is in the editor, the highlight bar in the edit window shows you where the compiler detected a problem. You can scroll up and down in the Transcript window to view the different messages.

Note that there is a “Compiling” message for each source file that was compiled. These messages serve as file boundaries, separating the various messages generated by each module and its include files. When you scroll to a message generated in a different source file, the edit window will only track in files that are currently loaded.

Thus, moving to a message that refers to an unloaded file causes the edit window’s highlight bar to turn off. Press Spacebar to load that file and continue tracking; the highlight bar will reappear. If you choose one of these messages (that is, press Enter when positioned on it), Borland C++ loads the file it references into an edit window and places the cursor on the error. If you then return to the Transcript window, tracking resumes in that file.

The Source Tracking settings in the Environment | Preferences subsection of the Settings notebook help you determine which window a file is loaded
Saving or deleting messages

You can use these settings when you’re message tracking and debug stepping.

Note that Previous message and Next message are affected by the Source Tracking setting. These commands will always find the next or previous error and will load the file using the method specified by the Source Tracking setting.

Normally, whenever you start to make a project, the Transcript window is cleared to make room for new messages. Sometimes, however, it is desirable to keep messages between makes.

Consider the following example: You have a project that has many source files and your program is set to stop on Errors. In this case, after compiling many files with warnings, one error in one file stops the make. You fix that error and want to find out if the fix works. But if you do a make or compile again, you lose your earlier warning messages. To avoid this, check Save Old Messages in the Environment | Preferences subsection of the Settings notebook. This way the only messages removed are the ones that result from the files you recompile. Thus, the old messages for a given file are replaced with any new messages that the compiler generates.

You can always get rid of all your messages by choosing Tools | Remove Messages, which deletes all the current messages. Unchecking Save Old Messages and running another make also gets rid of any old messages.

Autodependency checking

When you made your previous project, you dealt with the most basic situation: a list of C++ source file names. The Project Manager provides you with a lot of power to go beyond this simple situation.

The Project Manager collects autodependency information at compile time and caches these so that only files compiled outside the IDE need to be processed. The Project Manager can automatically check dependencies between source files in the project list (including files they themselves include) and their corresponding object files. This is useful when a particular C++ source file depends on other files. It is common for a C++ source to include several header files (.h files) that define the interface to external routines. If the interface to those routines changes, you’ll want the file that uses those routines to be recompiled.

If you’ve checked the Auto-Dependencies option in the Make section of the Settings notebook, Make obtains time-date stamps for all .CPP files and the files included by these. Then Make compares the date/time information of all these files with their date/time at last compile. If any date or time is different, the source file is recompiled.
If the Auto-Dependencies option is unchecked, the .CPP files are checked against .OBJ files. If earlier .CPP files exist, the source file is recompiled.

When a file is compiled, the IDE's compiler and the command-line compiler put dependency information into the .OBJ files. The Project Manager uses this to verify that every file that was used to build the .OBJ file is checked for time and date against the time and date information in the .OBJ file. The .CPP source file is recompiled if the dates are different.

That's all there is to dependencies. You get the power of more traditional makes while avoiding long dependency lists.

**Using different file translators**

So far you've built projects that use Borland C++ as the only language translator. Many projects consist of both C++ code and assembler code, and possibly code written in other languages. It would be nice to have some way to tell Borland C++ how to build such modules using the same dependency checks that we've just described. With the Project Manager, you don't need to worry about forgetting to rebuild those files when you change some of the source code, or about whether you've put them in the right directory, and so on.

For every source file that you have included in the list in the Project window, you can specify

- Which program (Borland C++, TASM, and so on) to use as its target file.
- Which command-line options to give that program.
- What to call the resulting module and where it will be placed (this information is used by the Project Manager to locate files needed for linking).
- Whether the module contains debug information.
- Whether the module gets included in the link.

By default, the IDE's compiler is chosen as the translator for each module, using no command-line local options, using the output directory for output, and assuming that debug information is not to be excluded.

Let's look at a simple example. Go to the Project window and move to the file MYFUNCS.CPP. Now press Ctrl+O to bring up the Local Options dialog box for this file.

Except for Borland C++, each of the names in the Project File Translators list box is a reference to a program defined in the Transfer section of the Settings notebook.
Press *Esc*, then *F10* to return to the main menu, then open the Settings notebook and turn to the Transfer section. The Transfer section contains a list of all the transfer programs currently defined. Use the arrow keys to select Turbo Assembler and press *Enter*. (Because the Edit button is the default, pressing *Enter* brings up the Modify/New Transfer Item dialog box.) Here you see that Turbo Assembler is defined as the program TASM in the current path. Notice that the Translator check box is marked with an X; this translator item is then displayed in the local Options dialog box. Press *Esc* to return to the Transfer section.

Suppose you want to compile the MYFUNCS module using the Borland C++ command-line compiler instead of the IDE's compiler. To do so, you would perform the following steps:

1. Define BCC as one of the Project File Translators in the Transfer dialog box. Move the cursor past the last entry in the Program Titles list, then press *Enter* to bring up the Modify/New Transfer Item dialog box. In the Program Title input box, type Borland C++ command-line compiler; in the Program Path input box, type $EDNAME. In the command line, type $EDNAME.

2. Check Translator by pressing *Spacebar* and press *Enter* (New is the default action button). Back at the Transfer dialog box, you see that Borland C++ command-line compiler is now in the Program Titles list box (the last part doesn't show). Choose OK and press *Enter*.

3. Back in the Project window, press *Ctrl+O* to go to the Local Options dialog box again. Notice that Borland C++ command-line compiler is now a choice on the Project File Translators list for MYFUNCS.CPP (as well as for all of your other files).

4. Tab to the Project File Translators list box and highlight Borland C++ command-line compiler (at this point, pressing *Enter* or tabbing to another group will choose this entry). Use the Command-line Options input box to add any command-line options you want to give BCＣ when compiling MYFUNCS.

MYFUNCS.CPP now compiles using BCC.EXE, while all of your other source modules compile with BC.EXE. The Project Manager applies the same criteria to MYFUNCS.CPP when deciding whether to recompile the module during a make as it does to all the modules that are compiled with BC.EXE.
Overriding libraries

In some cases, it's necessary to override the standard startup files or libraries. You override the startup file by placing a file called C0x.OBJ as the first name in your project file, where x stands for any name (for example, C0MINE.OBJ). It's critical that the name start with C0 and that it is the first file in your project.

To override the standard library, open the Settings notebook and turn to the Linker Settings subsection and, in the Standard Run-time Libraries box, select None for the Standard Run-time Library. Then add the library you want your project to use to the project file just as you would any other item.

More Project Manager features

Let's take a look at some of the other features the Project Manager has to offer. When you're working on a project that involves many source files, you want to be able to easily view portions of those files. You'll also want to be able to quickly access files that are included by others.

For example, expand MYMAIN.CPP to include a call to a function named GetMyTime:

```cpp
#include <iostream.h>
#include "myfuncs.h"
#include "mytime.h"

main(int argc, char *argv[]) {
    char *s;
    if (argc > 1)
        s=argv[1];
    else
        s="the universe";
    cout << GetString() << s << "\n";
}
```

This code adds one new include file to MYMAIN: mytime.h. Together myfuncs.h and mytime.h contain the prototypes that define the GetString and GetMyTime functions, which are called from MYMAIN. The mytime.h file contains

```cpp
#define HOUR 1
#define MINUTE 2
#define SECOND 3
extern int GetMyTime(int);
```
Go ahead and put the actual code for `GetMyTime` into a new source file called `MYTIME.CPP`:

```c
#include <time.h>
#include "mytime.h"

int GetMyTime(int which)
{
    struct tm *timeptr;
    time_t secsnow;

time(&secsnow);
timeptr = localtime(&secsnow);
switch (which) {
    case HOUR:
        return (timeptr -> tm_hour);
    case MINUTE:
        return (timeptr -> tm_min);
    case SECOND:
        return (timeptr -> tm_sec);
}
}
```

`MYTIME` includes the standard header file `time.h`, which contains the prototype of the `time` and `localtime` functions, and the definition of `tm` and `time_t`, among other things. It also includes `mytime.h` in order to define `HOUR`, `MINUTE`, and `SECOND`.

Create these new files, then use Project | Open Project to open `MYPROG.PRJ`. The files `MYMAIN.CPP` and `MYFUNCS.CPP` are still in the Project window. Now to build your expanded project, add the file name `MYTIME.CPP` to the Project window. Press `Ins` (or choose Project | Add Item) to bring up the Add Item dialog box. Use the dialog box to specify the name of the file you are adding and choose Done.

Now choose Compile | Make to make the project. `MYMAIN.CPP` will be recompiled because you’ve made changes to it since you last compiled it. `MYFUNCS.CPP` won’t be recompiled, because you haven’t made any changes to it since the make in the earlier example. `MYTIME.CPP` is compiled for the first time.

In the `MYPROG` project window, move to `MYMAIN.CPP` and press `Spacebar` (or Project | Include Files) to display the Include Files dialog box. This dialog box contains the name of the selected file, several buttons, and a list of include files and locations (paths). The first file in the Include Files list box is highlighted; the list box lists all the files that were included by the file `MYMAIN.CPP`. If any of the include files is located outside of the current directory, the path to the file is shown in the Location field of the list box.
As each source file is compiled, the information about which include files are included by which source files is stored in the source file’s .OBJ file. If you access the Include Files dialog box before you perform a make, it might contain no files or it might have files left over from a previous compile (which may be out of date). To load one of the include files into an edit window, highlight the file you want and press Enter or click the View button.

Looking at files in a project

Let's take a look at MYMAIN.CPP, one of the files in the Project. Simply choose the file using the arrow keys or the mouse, then press Enter. This brings up an edit window with MYMAIN.CPP loaded. Now you can make changes to the file, scroll through it, search for text, or whatever else you need to do. When you are finished with the file, save your changes if any, then close the edit window.

Suppose that after browsing around in MYMAIN.CPP, you realize that what you really wanted to do was look at mytime.h, one of the files that MYMAIN.CPP includes. Highlight MYMAIN.CPP in the Project window, then press Spacebar to bring up the Include Files dialog box for MYMAIN. (Alternatively, while MYMAIN.CPP is the active edit window, choose Project | Include Items. Now choose mytime.h in the Include Files box and press the View button. This brings up an edit window with mytime.h loaded. When you’re done, close the mytime.h edit window.
Command-line compiler

As an alternative to using the IDE, you can compile and run your programs with the command-line compiler (BCC.EXE). Almost anything you can do within the IDE can also be done using the command-line compiler. You can turn specific warnings on or off, invoke TASM (or another assembler) to assemble .ASM source files, invoke the linker to generate executable files, and so on.

This chapter is organized into two parts:

- The first part describes how to use the command-line compiler and provides a table of command-line compiler options along with a page-number cross-reference to where you can find detailed information about each option (see Table 6.1 starting on page 106).
- The second part, starting on page 111, presents the options organized functionally (with groups of related options).

To invoke Borland C++ from the command line, type BCC at the OS/2 shell prompt and follow it with a set of command-line arguments. Command-line arguments include compiler and linker options and file names. The generic command-line format is

```
BCC [option [option...]] filename [filename...]
```

Each command-line option must be preceded by either a hyphen (–) or slash (/), whichever you prefer. Each option must be separated by at least one space from the BCC command, other options, and file names that follow.

The options are divided into three general types:

- Compiler options, described starting on page 111.
- Linker options, described starting on page 124.
- Environment options, described starting on page 125.

To see an onscreen list of the options, type `BCC` (without any options or file names) at the OS/2 prompt, then press `Enter`. 
To select command-line options, enter a hyphen (-) or slash (/) immediately followed by the option letter (for example, -l or /l). To set an option off, add a second hyphen after the option letter. This is true for all toggle options (those that set an option on or off): A trailing hyphen (-) sets the option off, and a trailing plus sign (+) or nothing sets it on. So, for example, -C and -C+ both set nested comments on, while -C- sets nested comments off.

The option precedence rules are simple; command-line options are evaluated from left to right, and the following rules apply:

- For any option that is not an -I or -L option, a duplication on the right overrides the same option on the left. (Thus an off option on the right cancels an on option to the left.)
- The -I and -L options on the left, however, take precedence over those on the right.

Table 6.1: Command-line options summary

<table>
<thead>
<tr>
<th>Option</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@filename</td>
<td>110</td>
<td>Read compiler options from the response file filename.</td>
</tr>
<tr>
<td>+filename</td>
<td>110</td>
<td>Use the alternate configuration file filename.</td>
</tr>
<tr>
<td>-A</td>
<td>116</td>
<td>Use only ANSI keywords with strict compliance checking.</td>
</tr>
<tr>
<td>-AT</td>
<td>116</td>
<td>Use Borland C++ keywords (default).</td>
</tr>
<tr>
<td>-AK</td>
<td>117</td>
<td>Use only Kernighan and Ritchie keywords.</td>
</tr>
<tr>
<td>-AU</td>
<td>117</td>
<td>Use only UNIX keywords.</td>
</tr>
<tr>
<td>-an</td>
<td>113</td>
<td>Align to n: 1 = Byte, 2 = Word, 4 = Double Word boundaries.</td>
</tr>
<tr>
<td>-a-</td>
<td>113</td>
<td>Align byte (default).</td>
</tr>
<tr>
<td>-B</td>
<td>120</td>
<td>Compile and call the assembler to process inline assembly code.</td>
</tr>
<tr>
<td>-b</td>
<td>113</td>
<td>Make enums always int-sized (default).</td>
</tr>
<tr>
<td>-b-</td>
<td>113</td>
<td>Make enums byte-sized or word-sized when possible.</td>
</tr>
<tr>
<td>-C</td>
<td>117</td>
<td>Nested comments on.</td>
</tr>
<tr>
<td>-C-</td>
<td>117</td>
<td>Nested comments off (default).</td>
</tr>
<tr>
<td>-c</td>
<td>120</td>
<td>Compile to .OBJ but do not link.</td>
</tr>
<tr>
<td>-Dname</td>
<td>112</td>
<td>Define name to the null string.</td>
</tr>
<tr>
<td>-Dname=string</td>
<td>112</td>
<td>Define name to string.</td>
</tr>
<tr>
<td>-d</td>
<td>113</td>
<td>Merge duplicate strings on.</td>
</tr>
<tr>
<td>-d-</td>
<td>113</td>
<td>Merge duplicate strings off (default).</td>
</tr>
<tr>
<td>-ffilename</td>
<td>120</td>
<td>Use filename as the assembler to use.</td>
</tr>
<tr>
<td>-efilename</td>
<td>124</td>
<td>Link to produce filename.EXE.</td>
</tr>
<tr>
<td>-f</td>
<td>113</td>
<td>Fast floating point (default).</td>
</tr>
<tr>
<td>-f-</td>
<td>113</td>
<td>Strict ANSI floating point.</td>
</tr>
</tbody>
</table>
Table 6.1: Command-line options summary (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-G</td>
<td>Select code for speed.</td>
</tr>
<tr>
<td>-G-</td>
<td>Select code for size (default).</td>
</tr>
<tr>
<td>-gn</td>
<td>Warnings: stop after n messages.</td>
</tr>
<tr>
<td>-H</td>
<td>Causes the compiler to generate and use precompiled headers.</td>
</tr>
<tr>
<td>-H-</td>
<td>Turns off generation and use of precompiled headers (default).</td>
</tr>
<tr>
<td>-Hc</td>
<td>Cache precompiled headers. Must be used with -H or -Hxxx.</td>
</tr>
<tr>
<td>-Hu</td>
<td>Tells the compiler to use but not generate precompiled headers.</td>
</tr>
<tr>
<td>-H&quot;xxx&quot;</td>
<td>Stop compiling precompiled headers at file “xxx”. This must be used with -H, -Hu, or -H=filename.</td>
</tr>
<tr>
<td>-H=filename</td>
<td>Sets the name of the file for precompiled headers.</td>
</tr>
<tr>
<td>-Ipath</td>
<td>Directories for include files.</td>
</tr>
<tr>
<td>-in</td>
<td>Make significant identifier length to be n.</td>
</tr>
<tr>
<td>-Jg</td>
<td>Generate definitions for all template instances and merge duplicates (default).</td>
</tr>
<tr>
<td>-Jgd</td>
<td>Generate public definitions for all template instances; duplicates result in redefinition errors.</td>
</tr>
<tr>
<td>-Jgx</td>
<td>Generate external references for all template instances.</td>
</tr>
<tr>
<td>-jn</td>
<td>Errors: stop after n messages.</td>
</tr>
<tr>
<td>-K</td>
<td>Default character type unsigned.</td>
</tr>
<tr>
<td>-K-</td>
<td>Default character type signed (default).</td>
</tr>
<tr>
<td>-K2</td>
<td>Allow only two character types (unsigned and signed); char is treated as signed char.</td>
</tr>
<tr>
<td>-k</td>
<td>Standard stack frame on (default).</td>
</tr>
<tr>
<td>-k-</td>
<td>Standard stack frame off.</td>
</tr>
<tr>
<td>-Lpath</td>
<td>Directories for libraries.</td>
</tr>
<tr>
<td>-Ix</td>
<td>Pass option x to the linker (can use more than one x).</td>
</tr>
<tr>
<td>-I-x</td>
<td>Suppress option x for the linker.</td>
</tr>
<tr>
<td>-M</td>
<td>Instruct the linker to create a map file.</td>
</tr>
<tr>
<td>-N</td>
<td>Check for stack overflow.</td>
</tr>
<tr>
<td>-npath</td>
<td>Set the output directory.</td>
</tr>
<tr>
<td>-O2</td>
<td>Optimize for speed.</td>
</tr>
<tr>
<td>-O1</td>
<td>Optimize for size.</td>
</tr>
<tr>
<td>-Oa</td>
<td>Assume that pointer expressions are not aliased in common subexpression evaluation.</td>
</tr>
<tr>
<td>-Ob</td>
<td>Eliminate stores into dead variables.</td>
</tr>
<tr>
<td>-Oc</td>
<td>Enable local optimizations performed on blocks of code with single entry and single exit. The optimizations performed are common subexpression elimination, code reordering, branch optimizations, copy propagation, constant folding and code compaction.</td>
</tr>
<tr>
<td>-Od</td>
<td>Disable all optimizations, except jump distance optimization, which the compiler performs automatically.</td>
</tr>
<tr>
<td>-Oe</td>
<td>Enable global register allocation and data flow analysis.</td>
</tr>
<tr>
<td>-Oz</td>
<td>Enable all optimizations that perform transformations within an entire function, including global common subexpression elimination, loop invariant code motion, induction variable elimination, linear function test replacement, loop compaction and copy propagation.</td>
</tr>
<tr>
<td>-Oi</td>
<td>Enable inlining of intrinsic functions such as memcpy, strlen, and so on.</td>
</tr>
<tr>
<td>-Os</td>
<td>Attempts to minimize code size.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>-Ot</code></td>
<td>Attempts to maximize application execution speed.</td>
</tr>
<tr>
<td><code>-Os</code></td>
<td>Enables most speed optimizations (provided for Microsoft compatibility).</td>
</tr>
<tr>
<td><code>-ofilename</code></td>
<td>Compile source file to <code>filename.obj</code>.</td>
</tr>
<tr>
<td><code>-P</code></td>
<td>Perform a C++ compile regardless of source file extension.</td>
</tr>
<tr>
<td><code>-Pext</code></td>
<td>Perform a C++ compile and set the default extension to <code>ext</code>.</td>
</tr>
<tr>
<td><code>-P-</code></td>
<td>Perform a C++ or C compile depending on source file extension (default).</td>
</tr>
<tr>
<td><code>-P-ext</code></td>
<td>Perform a C++ or C compile depending on extension; set default extension to <code>ext</code>.</td>
</tr>
<tr>
<td><code>-p</code></td>
<td>Use Pascal (<code>_pascal</code>) calling convention.</td>
</tr>
<tr>
<td><code>-pc</code></td>
<td>Use C (<code>_cdecl</code>) calling convention.</td>
</tr>
<tr>
<td><code>-p-</code></td>
<td>Use register (<code>_fastcall</code>) calling convention for passing parameters in registers.</td>
</tr>
<tr>
<td><code>-r</code></td>
<td>Enable register variables (default).</td>
</tr>
<tr>
<td><code>-r-</code></td>
<td>Suppress the use of register variables.</td>
</tr>
<tr>
<td><code>-R</code></td>
<td>Include browser information in generated .OBJ files.</td>
</tr>
<tr>
<td><code>-RT</code></td>
<td>Enable run-time type information.</td>
</tr>
<tr>
<td><code>-S</code></td>
<td>Produce .ASM output file.</td>
</tr>
<tr>
<td><code>-sDfilename</code></td>
<td>Specify the name for the linker to use as the module definition file.</td>
</tr>
<tr>
<td><code>-sd</code></td>
<td>Link as a DLL.</td>
</tr>
<tr>
<td><code>-sm</code></td>
<td>Link with the multiple-thread libraries (*MT.LIB).</td>
</tr>
<tr>
<td><code>-Tstring</code></td>
<td>Pass <code>string</code> as an option to TASM or assembler specified with <code>-E</code>.</td>
</tr>
<tr>
<td><code>-Tr</code></td>
<td>Remove all previous assembler options.</td>
</tr>
<tr>
<td><code>-Uname</code></td>
<td>Undefined any previous definitions of <code>name</code>.</td>
</tr>
<tr>
<td><code>-u</code></td>
<td>Generate underscores (default).</td>
</tr>
<tr>
<td><code>-u-</code></td>
<td>Disables underscores.</td>
</tr>
<tr>
<td><code>-V</code></td>
<td>Smart C++ virtual tables.</td>
</tr>
<tr>
<td><code>-V0</code></td>
<td>External C++ virtual tables.</td>
</tr>
<tr>
<td><code>-V1</code></td>
<td>Public C++ virtual tables.</td>
</tr>
<tr>
<td><code>-Vmd</code></td>
<td>Use the smallest representation for member pointers.</td>
</tr>
<tr>
<td><code>-Vmm</code></td>
<td>Member pointers support multiple inheritance.</td>
</tr>
<tr>
<td><code>-Vmp</code></td>
<td>Honor the declared precision for all member pointer types.</td>
</tr>
<tr>
<td><code>-Vms</code></td>
<td>Member pointers support single inheritance.</td>
</tr>
<tr>
<td><code>-Vmv</code></td>
<td>Member pointers have no restrictions (most general representation).</td>
</tr>
<tr>
<td><code>-Vs</code></td>
<td>Local C++ virtual tables.</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Source debugging on.</td>
</tr>
<tr>
<td><code>-v-</code></td>
<td>Source debugging off.</td>
</tr>
<tr>
<td><code>-vi</code></td>
<td>Turns expansion of <code>inline</code> functions on.</td>
</tr>
<tr>
<td><code>-vi-</code></td>
<td>Turns expansion of <code>inline</code> functions off.</td>
</tr>
<tr>
<td><code>-w</code></td>
<td>Display warnings on.</td>
</tr>
<tr>
<td><code>-wxxx</code></td>
<td>Enable warning message <code>xxx</code>.</td>
</tr>
<tr>
<td><code>-w-xxx</code></td>
<td>Disable warning message <code>xxx</code>.</td>
</tr>
<tr>
<td><code>-X</code></td>
<td>Disable compiler autodependency output.</td>
</tr>
<tr>
<td><code>-x</code></td>
<td>Enable exception handling.</td>
</tr>
<tr>
<td><code>-xd</code></td>
<td>Enable destructor cleanup.</td>
</tr>
</tbody>
</table>
Table 6.1: Command-line options summary (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-xf</code></td>
<td>124 Expand function exception handling initialization inline.</td>
</tr>
<tr>
<td><code>-xp</code></td>
<td>124 Enable exception location information.</td>
</tr>
<tr>
<td><code>-y</code></td>
<td>115 Line numbers on.</td>
</tr>
<tr>
<td><code>-zAname</code></td>
<td>119 Code class.</td>
</tr>
<tr>
<td><code>-zBname</code></td>
<td>119 BSS class.</td>
</tr>
<tr>
<td><code>-zCname</code></td>
<td>119 Code segment.</td>
</tr>
<tr>
<td><code>-zDname</code></td>
<td>119 BSS segment.</td>
</tr>
<tr>
<td><code>-zEname</code></td>
<td>119 <code>_far16</code> segment.</td>
</tr>
<tr>
<td><code>-zFname</code></td>
<td>119 <code>_far16</code> class.</td>
</tr>
<tr>
<td><code>-zGname</code></td>
<td>119 BSS group.</td>
</tr>
<tr>
<td><code>-zHname</code></td>
<td>119 <code>_far16</code> group.</td>
</tr>
<tr>
<td><code>-zPname</code></td>
<td>119 Code group.</td>
</tr>
<tr>
<td><code>-zRname</code></td>
<td>120 Data segment.</td>
</tr>
<tr>
<td><code>-zSname</code></td>
<td>120 Data group.</td>
</tr>
<tr>
<td><code>-zTname</code></td>
<td>120 Data class.</td>
</tr>
<tr>
<td><code>-zVname</code></td>
<td>120 Far virtual table segment.</td>
</tr>
<tr>
<td><code>-zWname</code></td>
<td>120 Far virtual table.</td>
</tr>
<tr>
<td><code>-zX</code></td>
<td>120 Use default name for X (default).</td>
</tr>
</tbody>
</table>

---

**Syntax and file names**

C++ files have the extension .CPP; see page 121 for information on changing the default extension.

Borland C++ compiles files according to the following set of rules:

- `FILENAME.ASM` Invoke TASM to assemble to .OBJ.
- `FILENAME.OBJ` Include as object at link time.
- `FILENAME.LIB` Include as library at link time.
- `FILENAME` Compile `FILENAME.CPP`.
- `FILENAME.CPP` Compile `FILENAME.CPP`.
- `FILENAME.C` Compile `FILENAME.C`.
- `FILENAME.XYZ` Compile `FILENAME.XYZ`.

For example, suppose you have the following command line:

```
BCC -a -ff- -C -emyexe oldfile1 oldfile2 nextfile
```

Borland C++ compiles `OLDFILE1.CPP`, `OLDFILE2.CPP`, and `NEXTFILE.CPP` to an .OBJ, linking them to produce an executable program file named `MYEXE.EXE` with word alignment (`-a`), strict ANSI floating-point (`-ff-`), and nested comments (`-C`).
Borland C++ invokes TASM if you give it an .ASM file on the command line or if a .C or .CPP file contains inline assembly. Here are the options that the command-line compiler gives to TASM:

/D_ _LANG_ _/ml

LANG is CDECL, PASCAL, or STDCALL. The /ml option tells TASM to assemble with case sensitivity on.

Response files

Response files allow you to have longer command strings than OS/2 normally allows.

If you need to specify many options or files on the command line, you can place them in an ASCII text file, called a response file (you can, of course, name it anything you like). You can then tell the command-line compiler to read its command line from this file by including the appropriate file name prefixed with @. You can specify any number of such files, and you can mix them freely with other options and file names.

For example, suppose the file MOON.RSP contains STARS.C and RAIN.C. This command

BCC SUN.C @MOON.RSP ANYONE.C

causes Borland C++ to compile the files SUN.C, STARS.C, RAIN.C, and ANYONE.C. It expands to

BCC SUN.C STARS.C RAIN.C ANYONE.C

Any options included in a response file are evaluated just as if they had been typed in on the command line. See page 106 for a description of the rules for evaluating command-line options.

Configuration files

If you find you use a certain set of options over and over again, you can list them in a configuration file, called TURBOC.CFG by default. When you run BCC, it automatically looks for TURBOC.CFG in the current directory. If it doesn’t find it there, Borland C++ then looks in the startup directory (where BCC.EXE resides).

Remember that TURBOC.CFG is not the same as TCCONFIG.TC, which is the default IDE version of a configuration file.

You can create more than one configuration file; each must have a unique name. To specify the alternate configuration file name, include its file name, prefixed with +, anywhere on the BCC command line. For example, to read the option settings from the file C:\ALT.CFG, you could use the following command line:

BCC +C:\ALT.CFG ......
Your configuration file can be used in addition to or instead of options entered on the command line. If you don’t want to use certain options that are listed in your configuration file, you can override them with options on the command line.

You can create the TURBOC.CFG file (or any alternate configuration file) using any standard ASCII editor or word processor, such as Borland C++’s integrated editor. You can list options (separated by spaces) on the same line or list them on separate lines.

In general, you should remember that command-line options override configuration file options. If, for example, your configuration file contains several options, including the \texttt{-a} option (which you want to set off), you can still use the configuration file but override the \texttt{-a} option by listing \texttt{-a-} on the command line. However, the rules are a little more detailed than that. The option precedence rules detailed on page 106 apply, with these additional rules:

\begin{itemize}
  \item When the options from the configuration file are combined with the command-line options, any \texttt{-I} and \texttt{-L} options in the configuration file are appended to the right of the command-line options. This means that the include and library directories specified in the command line are the first ones that Borland C++ searches (thereby giving the command-line \texttt{-I} and \texttt{-L} directories priority over those in the configuration file).
  \item The remaining configuration file options are inserted immediately after the BCC command (to the left of any command-line options). This gives the command-line options priority over the configuration file options.
\end{itemize}

\section*{Compiler options}

Borland C++’s command-line compiler options fall into 11 groups; the page references to the left of each group tell where you can find a discussion of each kind of option:

\begin{itemize}
  \item See page 112. \textbullet{} Macro definitions let you define and undefine macros on the command line.
  \item See page 113. \textbullet{} Code-generation options govern characteristics of the generated code. Examples are the floating-point option, calling convention, character type, and CPU instructions.
  \item See Appendix A, “The optimizer.” \textbullet{} Optimization options let you specify how the object code is to be optimized.
\end{itemize}
See page 116.  ■ Source code options cause the compiler to recognize (or ignore) certain features of the source code: implementation-specific (non-ANSI, non-Kernighan and Ritchie, and non-UNIX) keywords, nested comments, and identifier lengths.

See page 117.  ■ Error-reporting options let you tailor which warning messages the compiler reports, and the maximum number of warnings and errors that can occur before the compilation stops.

See page 119.  ■ Segment-naming control options let you rename segments and reassign their groups and classes.

See page 120.  ■ Compilation control options let you direct the compiler to

- Compile to assembly code (rather than to an object module).
- Compile a source file that contains inline assembly.
- Compile without linking.
- Compile for PM applications.
- Use precompiled headers or not.

See page 121.  ■ C++ virtual table options let you control how virtual tables are handled.

See page 122.  ■ C++ member pointer options let you control how member pointers are used.

See page 123.  ■ Template generation options let you control how the compiler generates definitions or external declarations for template instances.

See page 124.  ■ Exception handling options let you selectively enable exception handling and generate runtime type identification.

---

**Macro definitions**

Macro definitions let you define and undefine macros (also called *manifest* or *symbolic* constants) on the command line. The default definition is the null string. Macros defined on the command line override those in your source file.

- **-Dname**  Defines the named identifier *name* to the null string.

- **-Dname=string**  Defines the named identifier *name* to the string *string* after the equal sign. *string* cannot contain any spaces or tabs.

- **-Uname**  Undefines any previous definitions of the named identifier *name*.

Borland C++ lets you make multiple `#define` entries on the command line in any of the following ways:
You can include multiple entries after a single `-D` option, separating entries with a semicolon (this is known as “ganging” options):

```
BCC -Dxxx;yyy=1;zzz=NO MYFILE.C
```

You can place more than one `-D` option on the command line:

```
BCC -Dxxx -Dyyy=1 -Dzzz=NO MYFILE.C
```

You can mix ganged and multiple `-D` listings:

```
BCC -Dxxx -Dyyy=1;zzz=NO MYFILE.C
```

Code-generation options govern characteristics of the generated code. Examples are the floating-point option, calling convention, character type, and CPU instructions.

`-a` Forces integer-size and larger items to be aligned on a machine-word boundary. Extra bytes are inserted in a structure to ensure member alignment. Automatic and global variables are aligned properly. `char` and `unsigned char` variables and fields can be placed at any address; all others are placed at an even-numbered address. This option is off by default (`-a-`), allowing bytewise alignment.

`-b` Tells the compiler to always allocate a four-byte `int` for enumeration types. This option is on by default.

`-b-` Tells the compiler to allocate the smallest integer that can hold the enumeration values. Thus, the compiler allocates an `unsigned` or `signed char` if the minimum and maximum values of the enumeration are both within the range of 0 to 255 or -128 to 127, respectively, or an `unsigned` or `signed short` if the minimum and maximum values of the enumeration are both within the range of 0 to 65,535 or -32,768 to 32,767, respectively. Otherwise the compiler uses a four-byte `int` to represent the enumeration values.

`-d` Tells the compiler to merge literal strings when one string matches another, thereby producing smaller programs. This can also cause errors if one string is modified while the other remains unmodified. This option is off by default (`-d-`). Setting on the `-d` option results in slightly longer compilation times.

`-ff` Tells the compiler to optimize floating-point operations without regard to explicit or implicit type conversions. Answers can be faster than under ANSI operating mode. See Chapter 10 in the *Programmer’s Guide* for details.

`-ff-` Sets off the fast floating-point option. The compiler follows strict ANSI rules regarding floating-point conversions.
-K Tells the compiler to treat all char declarations as if they were unsigned char type. This allows for compatibility with other compilers that treat char declarations as unsigned. By default, char declarations are signed (-K-).

-k Generates a standard stack frame, which is useful when using a debugger to trace back through the stack of called subroutines. This option is on by default. You can set it off by using option -k-.

-K2 Do not treat char as a distinct type. Treats char as signed char. Allows compatibility with Borland C++ 1.0. See the section on character constants in the Programmer's Guide, Chapter 1.

-N Generates stack overflow logic at the entry of each function. It causes a stack overflow message to appear when a stack overflow is detected. This is costly in terms of both program size and speed but is provided as an option because stack overflows can be very difficult to detect. If an overflow is detected, the message "Stack overflow!" is printed and the program exits with an exit code of 1.

-p Forces the compiler to generate all subroutine calls and all functions using the Pascal calling convention. This is equivalent to declaring all subroutine and functions with the __pascal keyword. The resulting function calls are usually smaller and faster than the -pc option. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments. You can use the __cdecl, __stdcall, or __fastcall keyword to specifically declare a function or subroutine using another calling convention.

-pc Forces the compiler to generate all subroutine calls and all functions using the Standard calling convention. This is equivalent to declaring all subroutine and functions with the __stdcall keyword. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments. You can use the __cdecl, __pascal, or __fastcall keyword to specifically declare a function or subroutine using another calling convention.

Forces the compiler to generate all subroutine calls and all functions using the C calling convention. This is equivalent to declaring all subroutine and functions with the __cdecl keyword. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments. You can use the __stdcall, __pascal, or __fastcall keyword to specifically declare a function or subroutine using another calling convention.
Unless you are an expert, don't use -u-. See Chapter 12, "Inline assembly," in the Programmer's Guide for details about underscores.

-pr

Forces the compiler to generate all subroutine calls and all functions using the Register calling convention. This is equivalent to declaring all subroutine and functions with the __fastcall keyword. The -pr often results in smaller and faster function calls. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments. You can use the __stdcall, __pascal, or __cdecl keyword to specifically declare a function or subroutine using another calling convention.

For more information about __fastcall, see Appendix A, "The optimizer."

-u

With -u selected, when you declare an identifier, Borland C++ automatically puts an underscore (_) in front of the identifier before saving the identifier in the object module.

Borland C++ treats Pascal identifiers (those modified by the __pascal keyword) differently—they are uppercase and are not prefixed with an underscore.

Underscores for C and C++ identifiers are optional, but are on by default. You can set them off with -u-. But note that setting the underscores off causes link errors when linking with the standard Borland C++ libraries.

-X

Disables generation of autodependency information in the output file. Modules compiled with this option enabled are not able to use the autodependency feature of MAKE or of the IDE. Normally this option is used only for files that are to be put into .LIB files (to save disk space).

-y

Includes line numbers in the object file for use by a symbolic debugger, such as Turbo Debugger. This increases the size of the object file but doesn’t affect size or speed of the executable program. This option is useful only in concert with a symbolic debugger that can use the information. In general, -v is more useful than -y with Turbo Debugger.

The -v and -vi options

-v

Tells the compiler to include debugging information in the .OBJ file so that the file(s) being compiled can be debugged with either Borland C++’s integrated debugger or the standalone Turbo
Turbo Debugger is both a source-level (symbolic) and assembly-level debugger. The compiler also passes this option on to the linker so it can include the debugging information in the .EXE file.

To facilitate debugging, this option also causes C++ inline functions to be treated as normal functions. To avoid that, use –vi.

- vi

Expands C++ inline functions inline.

In order to control the expansion of inline functions, the operation of the -v option is slightly different for C++. When inline function expansion is not enabled, the function is generated and called like any other function. Debugging in the presence of inline expansion can be extremely difficult, so Borland C++ provides the following options:

- v

Turns debugging on and inline expansion off.

- v-

Turns debugging off and inline expansion on.

- vi

Turns inline expansion on.

- vi-

Turns inline expansion off.

So, for example, if you want to turn both debugging and inline expansion on, you must use -v -vi.

- R

Includes browser information when the compiler generates .OBJ files; this lets you inspect the application while using the IDE's integrated Browser. When this option is off, you can link larger .OBJ files. This option doesn’t affect execution speed, but it does affect compile time.

Borland C++ is a professional optimizing compiler, featuring a number of options that let you specify how the object code is to be optimized; for size or speed, and using (or not) a wide range of specific optimization techniques. Appendix A, "The optimizer," discusses these options in detail.

Source code options cause the compiler to recognize (or ignore) certain features of the source code; implementation-specific (non-ANSI, non-Kernighan and Ritchie, and non-UNIX) keywords, nested comments, and identifier lengths. These options are most significant if you plan to port your code to other systems.

- A

Compiles ANSI-compatible code: Any of the Borland C++ extension keywords that are not prefixed with double underscores are ignored and can be used as normal identifiers. Note that C and C++ programs can use different keywords. The Programmer's Guide, Chapter 1, contains a complete discussion of keywords and register pseudovariables.
-A- Tells the compiler to use Borland C++ keywords. -AT is an alternate version of this option.

-AK Tells the compiler to use only Kernighan and Ritchie keywords.

-AT Tells the compiler to use Borland C++ keywords. -A- is an alternate version of this option.

-AU Tells the compiler to use only UNIX keywords.

-C Lets you nest comments. Comments normally cannot be nested (-C-).

-in Causes the compiler to recognize only the first n characters of identifier names. All identifiers, whether variables, preprocessor macros, or structure members, are treated as distinct only if their first n characters are distinct. Specifying n to be 0 or greater than 249, or not specifying the -in option at all, forces the compiler to allow identifiers of unlimited length.

By default, Borland C++ uses 32 characters per identifier. Other systems, including some UNIX compilers, ignore characters beyond the first eight. If you are porting to these other environments, you might want to compile your code with a smaller number of significant characters. Compiling in this manner helps you see if there are any name conflicts in long identifiers when they are truncated to a shorter significant length.

**Error-reporting options**

Error-reporting options let you tailor which warning messages the compiler reports, and the maximum number of warnings and errors that can occur before the compilation stops.

- gn Tells Borland C++ to stop compiling after n warning messages.

- jn Tells the compiler to stop compiling after n error messages.

- w Causes the compiler to display warning messages. You can set this off with -w-. You can enable or disable specific warning messages with -wxxx, described in the following paragraphs.

- wxxx Enables the specific warning message indicated by xxx. The option -w-xxx suppresses the warning message indicated by xxx. The possible options for -wxxx are divided into four categories: ANSI violations, frequent errors, portability warnings, and C++ warnings. Each category is discussed in the following sections. You can also use the pragma warn in your source code to control these options. See Chapter 5, "The preprocessor," in the Programmer's Guide.

For more information on these warnings, see Appendix A, "Error messages," in the Tools and Utilities Guide.
The asterisk (*) indicates that the option is on by default. All others are off by default.

ANSI violations
- `wbbf` Bit fields must be signed or unsigned int.
- `wbig` Hexadecimal value contains more than three digits.
- `wdpu` Declare type prior to use in prototype.
- `wdup` Redefinition of macro is not identical.
- `weas` Type assigned to enumeration.
- `wext` Identifier is declared as both external and static.
- `wpin` Initialization is only partially bracketed.
- `wret` Both return and return with a value are used.
- `wsus` Suspicious pointer conversion.
- `wvoi` Void functions cannot return a value.
- `wzdi` Division by zero.

Frequent errors
- `wamb` Ambiguous operators need parentheses.
- `wamp` Superfluous & with function or array.
- `wasm` Unknown assembler instruction.
- `waus` Identifier is assigned a value that is never used.
- `wccc` Condition is always true/false.
- `wdef` Possible use of identifier before definition.
- `weff` Code has no effect.
- `wll` Ill-formed pragma.
- `wnod` No declaration for function function.
- `wpai` Parameter parameter is never used.
- `wpia` Possibly incorrect assignment.
- `wpro` Call to function with no prototype.
- `wrch` Unreachable code.
- `wrvl` Function should return a value.
- `wstv` Structure passed by value.
- `wuse` Identifier is declared but never used.

Portability warnings
- `wcpt` Nonportable pointer comparison.
- `wrng` Constant out of range in comparison.
- `wret` Nonportable pointer conversion.
- `wsig` Conversion might lose significant digits.
- `wucp` Mixing pointers to signed and unsigned char.

C++ warnings
- `wbe` Initializing enumeration with type.
- `wdsz` Array size for ‘delete’ ignored.
- `whid` Function1 hides virtual function function2.
## Segment-naming control

Don't use these options unless you have a good understanding of segmentation on the 80386/80486 processor. Under normal circumstances, you do not need to specify segment names.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-wibc*</code></td>
<td>Base class <code>base1</code> is inaccessible because also in <code>base2</code>.</td>
</tr>
<tr>
<td><code>-winl*</code></td>
<td>Functions containing <code>identifier</code> are not expanded inline.</td>
</tr>
<tr>
<td><code>-wlin*</code></td>
<td>Temporary used to initialize <code>identifier</code>.</td>
</tr>
<tr>
<td><code>-wlvc*</code></td>
<td>Temporary used for parameter in call to <code>identifier</code>.</td>
</tr>
<tr>
<td><code>-wmpc*</code></td>
<td>Conversion to <code>type</code> fails for members of virtual base class <code>base</code>.</td>
</tr>
<tr>
<td><code>-wmpd*</code></td>
<td>Maximum precision used for member pointer type <code>type</code>.</td>
</tr>
<tr>
<td><code>-wncf*</code></td>
<td>Non-const function <code>function</code> called const object.</td>
</tr>
<tr>
<td><code>-wnci*</code></td>
<td>Constant member <code>identifier</code> is not initialized.</td>
</tr>
<tr>
<td><code>-wnst*</code></td>
<td>Use qualified name to access nested type <code>type</code>.</td>
</tr>
<tr>
<td><code>-wvtf*</code></td>
<td>Non-volatile function <code>function</code> called for volatile object.</td>
</tr>
<tr>
<td><code>-wobi*</code></td>
<td>Temporary used to initialize <code>identifier</code>.</td>
</tr>
<tr>
<td><code>-wofp*</code></td>
<td>Style of function definition is now obsolete.</td>
</tr>
<tr>
<td><code>-wovl*</code></td>
<td>Overload is now unnecessary and obsolete.</td>
</tr>
<tr>
<td><code>-wpre</code></td>
<td>Overloaded prefix operator <code>++/-</code> used as a postfix operator.</td>
</tr>
</tbody>
</table>

Segment-naming control options let you rename segments and reassign their groups and classes. See also the discussion of the preprocessor directive `codeseg` in the *Programmer's Guide*, Chapter 5.

- **-zAname** Changes the name of the code segment class to `name`. By default, the code segment is assigned to class CODE.

- **-zBname** Changes the name of the uninitialized data segment class to `name`. By default, the uninitialized data segments are assigned to class BSS.

- **-zCname** Changes the name of the code segment to `name`. By default, the code segment is named _TEXT.

- **-zDname** Changes the name of the uninitialized data segment to `name`. By default, the uninitialized data segment is named _BSS.

- **-zEname** Changes the name of the segment where `_far16` objects are put to `name`. By default, the segment name is the name of the source file followed by _DATA. A name beginning with an asterisk (*) indicates that the default string should be used.

- **-zFname** Changes the name of the class for `_far16` objects to `name`. By default, the name is DATA16. A name beginning with an asterisk (*) indicates that the default string should be used.

- **-zGname** Changes the name of the uninitialized data segment group to `name`. By default, the data group is named DGROUP.

- **-zHname** Causes `_far16` objects to be put into group `name`. By default, `_far16` objects are not put into a group. A name beginning...
Compilation control options with an asterisk (*) indicates that the default string should be used.

-\texttt{zPname}\vspace{2pt}
\begin{itemize}
\item Causes any output files to be generated with a code group for the code segment named \textit{name}.
\end{itemize}

-\texttt{zRname}\vspace{2pt}
\begin{itemize}
\item Sets the name of the initialized data segment to \textit{name}. By default, the initialized data segment is named \_DATA.
\end{itemize}

-\texttt{zSname}\vspace{2pt}
\begin{itemize}
\item Changes the name of the initialized data segment group to \textit{name}. By default, the data group is named DGROUP.
\end{itemize}

-\texttt{zTname}\vspace{2pt}
\begin{itemize}
\item Sets the name of the initialized data segment class to \textit{name}. By default, the initialized data segment class is named DATA.
\end{itemize}

-\texttt{zVname}\vspace{2pt}
\begin{itemize}
\item Sets the name of the far virtual table segment to \textit{name}. By default, far virtual tables are generated in the code segment.
\end{itemize}

-\texttt{zWname}\vspace{2pt}
\begin{itemize}
\item Sets the name of the far virtual table class segment to \textit{name}. By default, far virtual table classes are generated in the CODE segment.
\end{itemize}

-\texttt{zX*}\vspace{2pt}
\begin{itemize}
\item Uses the default name for X. For example, \texttt{-zA*} assigns the default class name CODE to the code segment.
\end{itemize}

Compilation control options let you control compilation of source files, such as whether your code is compiled as C or C++, whether to use precompiled headers, and what kind of PM executable file is created. For more detailed information on how to create a PM application, see Chapter 9, “Building OS/2 applications” in the Programmer’s Guide.

\begin{itemize}
\item \texttt{-B}\vspace{2pt}
\begin{itemize}
\item Compiles and calls the assembler to process inline assembly code.
\end{itemize}
\item \texttt{-c}\vspace{2pt}
\begin{itemize}
\item Compiles and assembles the named .C, .CPP, and .ASM files, but does not execute a link command.
\end{itemize}
\item \texttt{-Ef\textsl{filename}}\vspace{2pt}
\begin{itemize}
\item Uses \textit{name} as the name of the assembler to use. By default, TASM is used.
\end{itemize}
\item \texttt{-H}\vspace{2pt}
\begin{itemize}
\item Causes the compiler to generate and use precompiled headers, using the default filename BCDEF.CSM.
\end{itemize}
\item \texttt{-H-}\vspace{2pt}
\begin{itemize}
\item Sets generation and use of precompiled headers off (this is the default). Precompiled headers can dramatically increase compile speed, although they require considerable disk space.
\end{itemize}
\item \texttt{-Hc}\vspace{2pt}
\begin{itemize}
\item Cache precompiled headers. This option requires the use of \texttt{-H} or \texttt{-Hxxx}.
\end{itemize}
\end{itemize}
-Hu  Tells the compiler to use but not generate precompiled headers.

-H"xxx"  Stop compiling precompiled headers at file "xxx". This option requires the use of -H, -Hu, or -H=filename.

-H=filename  Sets the name of the file for precompiled headers, if you want to save this information in a file other than BCDEF.CSM. This option also sets on generation and use of precompiled headers; that is, it also has the effect of -H.

-o filename  Compiles the named file to the specified filename.obj.

-P  Causes the compiler to compile your code as C++ always, regardless of extension. The compiler assumes that all files have .CPP extensions unless a different extension is specified with the code.

-Pext  Causes the compiler to compile all files as C++; it changes the default extension to whatever you specify with ext. This option is available because some programmers use .C or another extension as their default extension for C++ code.

-P-  Tells the compiler to compile a file as either C or C++, based on its extension. The default extension is .CPP. This option is the default.

-P-ext  Tells the compiler to compile code based on the extension (.CPP as C++ code, all other file-name extensions as C code). It further specifies what the default extension is to be.

-S  Compiles the named source files and produces assembly language output files (.ASM), but does not assemble. When you use this option, Borland C++ includes the C or C++ source lines as comments in the produced .ASM file.

-T string  Passes string as an option to TASM (or as an option to the assembler defined with -E).

-T-  Removes all previously defined assembler options.

---

**C++ virtual tables**

The -V option controls the C++ virtual tables. It has four variations:

-Use this option when you want to generate common C++ virtual tables and out-of-line **inline** functions across modules within your application. As a result, only one instance of a given virtual table or out-of-line **inline** function is included in the program. This produces the smallest and most efficient executables.
Use this option when you want to generate local virtual tables and out-of-line inline functions. As a result, each module gets its own private copy of each virtual table or out-of-line inline function it uses; this setting produces larger executables than the Smart setting.

These options work together to create global virtual tables. If you don’t want to use the Smart or Local options (-V or -Vs), you can use -V0 and -V1 to produce and reference global virtual tables. -V0 generates external references to virtual tables; -V1 produces public definitions for virtual tables.

When using these two options, at least one of the modules in the program must be compiled with the -V1 option to supply the definitions for the virtual tables. All other modules should be compiled with the -V0 option to refer to that Public copy of the virtual tables.

The Borland C++ compiler supports several different kinds of member pointer types, with varying degrees of complexity and generality. By default, the compiler uses the most general (but in some contexts also the least efficient) kind for all member pointer types; this default behavior can be changed via the -Vm family of switches.

Member pointers declared while this option is in effect have no restriction on what members they can point to; they use the most general representation.

Member pointers declared while this option is in effect are allowed to point to members of multiple inheritance classes, except that members of virtual base classes cannot be pointed to.

Member pointers declared while this option is in effect are not allowed to point to members of classes that are base classes of classes with multiple inheritance (in general, they can be used with single inheritance classes only).

Member pointers declared while this option is in effect use the smallest possible representation that allows member pointers to point to all members of their class. If the class is not fully defined at the point where the member pointer type is declared, the most general representation has to be chosen by the compiler (and a warning is issued about this).

Whenever a member pointer is dereferenced or called, the compiler treats the member pointer as if it were of the least general case needed for that particular pointer type. For
example, a call through a pointer to a member of a class that is declared without any base classes treats the member pointer as having the simplest representation, regardless of how it’s been declared. This works correctly (and produces the most efficient code) in all cases except for one: when a pointer to a derived class is explicitly cast as a pointer-to-member of a ‘simpler’ base class, when the pointer is actually pointing to a derived class member. This is a non-portable (and dubious) construct, but if you need to compile code that uses it, use the `-Vmp` option. This forces the compiler to honor the declared precision for all member pointer types.

<table>
<thead>
<tr>
<th>Template generation options</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>-Jg</code> option controls the generation of template instances in C++. It has three variations:</td>
</tr>
<tr>
<td><strong>-Jg</strong></td>
</tr>
<tr>
<td><strong>-Jgd</strong></td>
</tr>
<tr>
<td><strong>-Jgx</strong></td>
</tr>
</tbody>
</table>

For more information about templates, see Chapter 3, “C++ specifics,” in the Programmer’s Guide.
**Exception handling/RTTI**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-x</code></td>
<td>Enables C++ exception handling. If you use C++ exception handling constructs in your code and compile with this option disabled (by unchecking the option in the IDE or using the <code>-x</code> command-line option), you'll get an error. See also the Library Reference, Chapter 9, for a description of <code>set_new_handler</code> function.</td>
</tr>
<tr>
<td><code>-xp</code></td>
<td>Enables exception location information that makes available run-time identification of exceptions by providing the line numbers in the source code where the exception occurred. This lets the program query the file and line number from where a C++ exception occurred.</td>
</tr>
<tr>
<td><code>-xd</code></td>
<td>Enables destructor cleanup so that destructors are called for all automatically declared objects between the scope of the catch and throw statements when an exception is thrown. Note that destructors aren't automatically called for dynamic objects and dynamic objects aren't automatically freed.</td>
</tr>
<tr>
<td><code>-xf</code></td>
<td>Normally, the prolog of a function with any exception handling constructs will contain a call to a run-time library function to initialize exception handling for the function. The <code>-xf</code> option expands this code inline in the prolog of each function providing slightly faster code execution, but at the expense of code size. This option can be used selectively for only the most time-critical functions.</td>
</tr>
<tr>
<td><code>-RT</code></td>
<td>Enables runtime type identification (RTTI). This is on by default.</td>
</tr>
<tr>
<td><code>-RT-</code></td>
<td>Turns the default RTTI option off.</td>
</tr>
</tbody>
</table>

**Linker options**

`-efilename` Derives the executable program's name from `filename` by adding the file extension .EXE (the program name is then `filename.EXE`). `filename` must immediately follow the `-e`, with no intervening whitespace. Without this option, the linker derives the .EXE file's name from the name of the first source or object file in the file name list. The default extension is .DLL when you are using `-sd`. 

`-lx` Passes option `x` to the linker. More than one option can appear after the `-i` (which is a lowercase l), with each option separated by a semicolon.
-l-x Suppresses linker option x. More than one option can appear after the -l- (lowercase l followed by a dash), with each option separated by a semicolon.

-M Forces the linker to produce a full link map. The default is to produce no link map.

-sDfilename Forces the linker to use filename as the module definition file.

-sd Produces a DLL file.

-sm Links with the OS/2 multi-thread libraries. This also defines the macro __MT__.

Environment options

When working with environment options, bear in mind that Borland C++ recognizes two types of library files: implicit and user-specified (also known as explicit library files). These are defined and discussed on page 126.

-Ilpath Causes the compiler to search path (the drive specifier or path name of a subdirectory) for include files (in addition to searching the standard places). A drive specifier is a single letter, either uppercase or lowercase, followed by a colon (:). A directory is any valid directory or directory path. You can use more than one -I (which is an uppercase I) directory option.

-Lpath Forces the linker to get the C0x.OBJ start-up object file and the Borland C++ library files (Cx.LIB, CxMT.LIB, and OS2.LIB) from the named directory. By default, the linker looks for them in the current directory.

-npath Places any .OBJ or .ASM files created by the compiler in the directory or drive named by path.

Borland C++ can search multiple directories for include and library files. This means that the syntax for the library directories (-L) and include directories (-I) command-line options, like that of the #define option (-D), allows multiple listings of a given option.

Here is the syntax for these options:

Library directories: -Ldirname[;dirname;...] 
Include directories: -Idirname[;dirname;...]
The parameter dirname used with -L and -I can be any directory or directory path.

You can enter these multiple directories on the command line in the following ways:
- You can "gang" multiple entries with a single -L or -I option, separating ganged entries with a semicolon, like this:
  BCC -Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc3 myfile.c
- You can place more than one of each option on the command line, like this:
  BCC -Ldirname1 -Ldirname2 -Ldirname3 -Iinc1 -Iinc2 -Iinc3 myfile.c
- You can mix ganged and multiple listings, like this:
  BCC -Ldirname1;dirname2 -Ldirname3 -Iinc1;inc2 -Iinc3 myfile.c

If you list multiple -L or -I options on the command line, the result is cumulative: The compiler searches all the directories listed, in order from left to right.

**Note**
The IDE also supports multiple library directories through the "ganged entry" syntax.

The Borland C++ include-file search algorithms search for the header files listed in your source code in the following way:
- If you put an #include <somefile.h> statement in your source code, Borland C++ searches for somefile.h only in the specified include directories.
- If, on the other hand, you put an #include "somefile.h" statement in your code, Borland C++ searches for somefile.h first in the current directory; if it does not find the header file there, it then searches in the include directories specified in the command line.

The library file search algorithms are similar to those for include files:
- Implicit libraries: Borland C++ searches for implicit libraries only in the specified library directories; this is similar to the search algorithm for #include <somefile.h>. Implicit library files are the ones Borland C++ automatically links in, such as Cx.LIB, OS2.LIB, and the start-up object file (C0x.OBJ).
- Explicit libraries: Where Borland C++ searches for explicit (user-specified) libraries depends in part on how you list the library file name. Explicit library files are ones you list on the command line or in a project file; these are file names with a .LIB extension.
• If you list an explicit library file name with no drive or directory (like this: mylib.lib), Borland C++ searches for that library in the current directory first. Then (if the first search was unsuccessful), it looks in the specified library directories. This is similar to the search algorithm for \#include "somefile.h".

• If you list a user-specified library with drive and/or directory information (like this: c:\mystuff\mylib1.lib), Borland C++ searches only in the location you explicitly listed as part of the library path name and not in the specified library directories.

Here is an example of how to compose a Borland C++ command line for an application that uses special header and library files.

1. Your current drive is C:, and your current directory is C:\BCOS2, where your source code resides. Your A drive’s current directory is A:\ASTROLIB.
2. Your include files (.h or “header” files) are located in C:\BCOS2\INCLUDE.
3. Your startup files (C02.OBJ, C02D.OBJ, and so forth) are in C:\BCOS2\LIB.
4. Your standard Borland C++ library files (C2.LIB, C2MT.LIB, ..., OS2.LIB, and so forth) are in C:\BCOS2\LIB.
5. Your custom library files for star systems (which you created and manage with TLIB) are in C:\BCOS2\STARLIB. One of these libraries is PARX.LIB.
6. Your third-party-generated library files for quasars are in the A drive in \ASTROLIB. One of these libraries is WARP.LIB.

Under this configuration, you enter the following command:

```
BCC -Llib;starlib -linclude orion.c umaj.c parx.lib a:\astrolib\warp.lib
```

Borland C++ compiles ORION.C and UMAJ.C to .OBJ files, searching C:\BCOS2\INCLUDE for any header files in your source code. It then links ORION.OBJ and UMAJ.OBJ with the start-up code (C02.OBJ), the standard libraries (C2.LIB and OS2.LIB), and the user-specified libraries (PARX.LIB and WARP.LIB), producing an executable file named ORION.EXE.

It searches for the startup code in C:\BCOS2\LIB (then stops because it’s there); it searches for the standard libraries in C:\BCOS2\LIB (and stops because they’re there).

When it searches for the user-specified library PARX.LIB, the compiler first looks in the current directory, C:\BCOS2. Not finding the library there, the
compiler then searches the library directories in order: first C:\BCOS2\LIB, then C:\BCOS2\STARLIB (where it locates PARX.LIB).

Because an explicit path is given for the library WARP.LIB (A:\ASTROLIB\WARP.LIB), the compiler only looks there.
The optimizer

This appendix details the use of the Borland C++ optimization options, including command-line options and IDE settings.

What is optimization?

Borland C++ is a professional optimizing compiler that gives you complete control over what kinds of optimization you want the compiler to perform.

An optimizer is a tool for improving your application’s speed or reducing its size. Borland’s optimizer provides extensive state-of-the-art optimization technology, providing a boost in speed or a reduction in size without affecting the style in which you like to program.

You can use the optimizer from the earliest stage of development to the final stages without having to worry about slow compilation times. Although most compilers take two to three times longer to compile when performing full optimizations, the Borland C++ compiler takes only 60% longer. In addition, the Borland C++ debugger understands optimized code, so debugging your optimized application is easy.

Optimization options

The command-line compiler controls most optimizations through the -O command-line option. The -O option can be followed by one or more of the suboption letters given in the list below. For example, -Oaxt turns on all speed optimizations and the Assume No Pointer Aliasing optimization. You can turn off optimizations on the command line by placing a minus before the optimization letter. For example, -O2-z turns on all speed optimizations except the global transformation optimizations. In addition, some optimizations are controlled by means other than -O. For example, the -r option enables the use of register variables.
The optimizations options follow the same rules for precedence as all other Borland C++ options. For example, `-Od` appearing on the command line after a `-O2` disables all optimizations.

The settings shown for each optimization in table A.1 are located in the Settings notebook. To access the Settings notebook, choose the View Settings option from the Project menu. For information on the Settings notebook, see Chapter 4, “Settings notebook.”

<table>
<thead>
<tr>
<th>Command-line</th>
<th>IDE Setting and Optimization Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-O2</code></td>
<td>Compiler[Optimizations]Fastest Code</td>
</tr>
<tr>
<td></td>
<td>Generates the fastest code possible. This is the same as using the following command-line options: <code>-O -Ob -Oe -Oz -Oi -Ot -Oc</code>.</td>
</tr>
<tr>
<td><code>-O1</code></td>
<td>Compiler[Optimizations]Smallest Code</td>
</tr>
<tr>
<td></td>
<td>Generates the smallest code possible. This is the same as using the following command-line options: <code>-O -Ob -Os -Oc -Oe</code>.</td>
</tr>
<tr>
<td><code>-Oa</code></td>
<td>Compiler[Optimizations]Assume No Pointer Aliasing</td>
</tr>
<tr>
<td></td>
<td>Assume that pointer expressions are not aliased in common subexpression evaluation.</td>
</tr>
<tr>
<td><code>-Ob</code></td>
<td>Compiler[Optimizations]Dead Storage Elimination</td>
</tr>
<tr>
<td></td>
<td>Eliminates dead variables.</td>
</tr>
<tr>
<td><code>-Oc</code></td>
<td>Compiler[Optimizations]Local Common Expressions</td>
</tr>
<tr>
<td></td>
<td>Enables local optimizations that are performed on blocks of code that have single entry and single exit. The optimizations performed are common subexpression elimination, code reordering, branch optimizations, copy propagation, constant folding and code compaction.</td>
</tr>
<tr>
<td><code>-Od</code></td>
<td>Compiler[Optimizations]Minimal Opts</td>
</tr>
<tr>
<td></td>
<td>Disables all optimizations, except jump distance optimization, which the compiler performs automatically.</td>
</tr>
<tr>
<td><code>-Oe</code></td>
<td>Compiler[Optimizations]Global Register Allocation</td>
</tr>
<tr>
<td></td>
<td>Enables global register allocation and data flow analysis.</td>
</tr>
<tr>
<td><code>-Oi</code></td>
<td>Compiler[Optimizations]Intrinsic Expansion</td>
</tr>
<tr>
<td></td>
<td>Enables inlining of intrinsic functions such as <code>memcpy</code>, <code>strlen</code>, and so on.</td>
</tr>
<tr>
<td><code>-Os</code></td>
<td>Compiler[Optimizations]Optimize For</td>
</tr>
<tr>
<td></td>
<td>Attempts to minimize code size.</td>
</tr>
<tr>
<td><code>-Ot</code></td>
<td>Compiler[Optimizations]Optimize For</td>
</tr>
<tr>
<td></td>
<td>Attempts to maximize application execution speed.</td>
</tr>
<tr>
<td><code>-Ox</code></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Enables most speed optimizations. This is provided for compatibility with Microsoft compilers.</td>
</tr>
<tr>
<td><code>-Oz</code></td>
<td>Compiler[Optimizations]Global Optimizations</td>
</tr>
<tr>
<td></td>
<td>Enables all optimizations that perform transformations within an entire function. They are: global common subexpression elimination, loop invariant code motion, induction variable elimination, linear function test replacement, loop compaction and copy propagation.</td>
</tr>
</tbody>
</table>
Table A.1: Optimization options summary (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-r</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>This option enables the use of register variables. It is on by default.</td>
</tr>
<tr>
<td>-r-</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>This option suppresses the use of register variables. When you are using this option, the compiler won't use register variables, and it won't preserve and respect register variables (ESI, EDI, and EBX) from any caller. For that reason, you should not have code that uses register variables call code which has been compiled with -r-. On the other hand, if you are interfacing with existing assembly-language code that does not preserve ESI, EDI, and EBX, the -r- option allows you to call that code from Borland C++.</td>
</tr>
</tbody>
</table>

A closer look at the Borland C++ optimizer

The Borland C++ optimizer performs a number of optimizations, including sophisticated register coloring, invariant code motion, induction variable elimination, and many others. Each of these optimizations has been fine-tuned to the complex instruction set of the Intel 80x86. In addition, the compiler performs architecture-specific optimizations for the target processor. The following sections describe these optimizations.

Global register allocation

Because memory references are so expensive on the 80x86 processors, it is extremely important to minimize those references through the intelligent use of registers. Global register allocation both increases the speed and decreases the size of your application. You should always use global register allocation when compiling your application with optimizations on.

Global optimizations

The Borland C++ compiler is designed to provide the most efficient code possible with the minimum increase in compilation speed. Thus, a number of optimizations are grouped together and performed in a single step. These optimizations are global common subexpression elimination, invariant code motion, induction variable elimination, copy propagation, loop compaction and linear function test replacement. Because all these optimizations are performed in a single step, you can’t set any of them on or off individually. You can set them all on with the -Oz option, or set them all off with the -Oz- option.

Common subexpression elimination

Common subexpression elimination is the process of finding duplicate expressions within the target scope and eliminating the duplicate expression by using the value of the previous expression it had computed. This avoids having to recalculate the expression. When you use this optimization in conjunction with global register allocation, the gains are both in size reduction and speed increase; otherwise, the gain is mainly a speed increase. Common subexpression elimination lets you program in a
more readable style, without the need to create unnecessary temporary locations for expressions that are used more than once. For example, the following code uses a temporary variable to avoid using expensive pointer referencing:

```c
temp = t->n.o.left;
if(temp->op == O_ICON || temp->op == O_FCON)
    ;
```

With common subexpression elimination, you can use direct referencing, which is more readable and easier to understand, and let the optimizer decide whether it is more efficient to create the temporary variable.

```c
if(t->n.o.left->op == O_ICON || t->n.o.left->op == O_FCON)
    ;
```

Moving invariant code out of loops is a speed optimization. The optimizer uses the information about all the expressions in the function gathered during data flow analysis to find expressions whose values do not change inside a loop. To prevent the calculation from being performed many times inside the loop, the optimizer moves the code outside the loop so that it is calculated only once. The optimizer then reuses the calculated value inside the loop. For example, in the code below, \(x \times y \times z\) is evaluated in every iteration of the loop.

```c
int v[10];
void f(void) {
    int i, x, y, z;
    for (i = 0; i < 10; i++)
        v[i] = v[i] * x * y * z;
}
```

The optimizer rewrites the code for the loop so that it looks like this:

```c
int v[10];
void f(void) {
    int i, x, y, z, t1;
    t1 = x * y * z;
    for (i = 0; i < 10; i++)
        v[i] = v[i] * t1;
}
```

Copy propagation is primarily a speed optimization. Like loop invariant code motion, copy propagation relies on the data flow analysis. The optimizer remembers the values assigned to expressions and uses those values instead of loading the value of the assigned expressions. Copies of
constants, expressions, and variables may be propagated. For example, in the following code the constant value 5 can be used for the second assignment instead of the expression on the right side, so that:

```c
PtrParIn->IntComp = 5;
{ *(PtrParIn->PtrComp) }.IntComp = PtrParIn->IntComp;
```

is optimized to look like:

```c
{ *(PtrParIn->PtrComp) }.IntComp = PtrParIn->IntComp = 5;
```

Induction variable analysis and strength reduction are speed optimizations performed on loops. The optimizer uses a mathematical technique called induction to create new variables out of expressions used inside a loop. These variables are called induction variables. The optimizer assures that the operations performed on these new variables are computationally less expensive (reduced in strength) than those used by the original variables.

Opportunities for these optimizations are common if you use array indexing or structure references inside loops, where these references vary with the loop iterations. For example, the optimizer creates an induction variable out of the operation \( v[i] \) in the code below, because the \( v[i] \) operation varies with the iterative nature of the loop.

```c
int v[10];
void f(void) {
  int i, x, y, z;
  for (i = 0; i < 10; i++)
    v[i] = x * y * z;
}
```

The optimizer changes this code to the following:

```c
int v[10];
void f(void) {
  int i, x, y, z, *p;
  p = v;
  for (i = 0; i < 10; i++)
    *p = x * y * z;
  p++;
}
```

Linear function test replacement is an optimization that occurs when induction variable elimination has taken place. Induction variable elimination generates expressions that vary linearly with the loop iterations. The compiler can replace the test condition of the loop with an induction variable expression and scale the test operands accordingly. This
optimization is done when the loop iterator varies linearly and is not used
directly within the loop and if its value is not required outside the loop. For
example, the loop iterator i is used only to count the for loop, and is not
used outside the for loop.

```c
int v[10];
void f(void) {
    int i, x, y, z, *p;
    p = v;
    for (i = 0; i < 10; i++)
        *p = x * y * z;
    p++;
}
```

After being optimized, the code looks like this:

```c
int v[10];
void f(void) {
    int i, x, y, z, *p;
    for (p = v; p < &v[10]; p++)
        *p = x * y * z;
}
```

This eliminates the need for the loop iterator i.

Loop compaction takes advantage of the string move instructions on the
80x86 processors by replacing the code for a loop with such an instruction.

```c
int v[100];
void t(void) {
    int i;
    for (i = 0; i < 100; i++)
        v[i] = 0;
}
```

The optimizer reduces this to the machine instructions:

```
mov  ecx,100
mov  edi,offset _v[0]
xor  eax, eax
rep  stosd
```

Depending on the complexity of the operands, the compacted loop code
might also be smaller than the corresponding non-compacted loop. You
might want to experiment with this optimization if you are compiling for
size and have loops of this nature.
The optimizer can identify variables that are no longer needed or that are unnecessary. In the following example, the optimizer performs induction variable elimination and linear function test replacement to reveal a dead loop iterator \( j \). Using -Ob removes the code to store any result into variable \( j \).

```c
int goo(void), a[10];

int f(void) {
    int i, j;
    i = goo();
    for(j = 0; j < 10; j++)
        a[j] = goo();
    return i;
}
```

After the dead storage elimination optimization is performed on this code, it looks like this:

```c
int goo(void), a[10];

int f(void) {
    int i;
    i = goo(); // The 'j = ' has been removed.
    for(int *p = &a[0]; p < &a[10]; p++)
        *p = goo();
    return i;
}
```

Pointer aliasing is not an optimization in itself, but it does affect optimizer performance. Since C and C++ allow pointers to point to any type, the compiler normally gathers pointer information to generate clean, correct code. When a pointer has global scope, the compiler is not able to determine what it points to, and takes the conservative view that it could point to every variable that is in global scope. This might be too conservative for your program. Pointer aliasing provides a mechanism by which you can inform the compiler that such cases do not exist and that two pointers do not point to the same location, thus allowing the compiler to be more aggressive and generate better code. Pointer aliasing might create bugs, which are hard to spot, so it is only applied when you use -Oa.

-Oa controls how the optimizer treats expressions with pointers in them. When compiling with global or local common subexpressions and -Oa enabled, the optimizer recognizes

\[ *p * x \]
as a common subexpression in function foo in the following code:

```c
int g, y;
int foo(int *p) {
    int x=5;
    y = *p * x;
    g = 3;
    return (*p * x);
}
void goo(void) {
    g=2;
    foo(&g);  /* This is incorrect, because the
               assignment g = 3 invalidates the
               expression *p * x. */
}
```

-Oa also controls how the optimizer treats expressions involving variables whose address has been taken. When compiling with -Oa, the compiler assumes that assignments via pointers affect only those expressions involving variables whose addresses have been taken and which are of the same type as the left-hand side of the assignment in question. To illustrate, consider the following function:

```c
int y, z;
int f(void) {
    int x;
    char *p = (char *)&x;
    y = x * z;
    *p = 'a';
    return (x * z);
}
```

When compiled with -Oa, the assignment *p = 'a' does not prevent the optimizer from treating x * z as a common subexpression, because the destination of the assignment, *p, is a char, whereas the addressed variable is an int. When compiled without -Oa, the assignment to *p prevents the optimizer from creating a common subexpression out of x * z.

You can control the selection and compaction of instructions with the -G and -G- options. -G tells the compiler to compile your source code for the fastest execution time. This is equivalent to pressing the Fastest Code button in the Compiler | Optimizations subsection of the Settings notebook.
There are times when you might want to use one of the common string or memory functions, such as `strcpy` or `memcmp`, but you don’t want to incur the overhead of a function call. If you use `-Oi`, the compiler generates the code for these functions within your function’s scope, eliminating the need for a function call. The resulting code executes faster than a call to the same function, but it is also larger.

The following is a list of those functions that are inlined when `-Oi` is enabled.

<table>
<thead>
<tr>
<th>Function 1</th>
<th>Function 2</th>
<th>Function 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alloca</code></td>
<td><code>memset</code></td>
<td><code>strcmp</code></td>
</tr>
<tr>
<td><code>fabs</code></td>
<td><code>_rotr</code></td>
<td><code>strcpy</code></td>
</tr>
<tr>
<td><code>_lrotl</code></td>
<td><code>_rotr</code></td>
<td><code>strlen</code></td>
</tr>
<tr>
<td><code>_lror</code></td>
<td><code>_rrotl</code></td>
<td><code>strncat</code></td>
</tr>
<tr>
<td><code>memchr</code></td>
<td><code>_rrotl</code></td>
<td><code>strncmp</code></td>
</tr>
<tr>
<td><code>memcpy</code></td>
<td><code>stpcpy</code></td>
<td><code>strncpy</code></td>
</tr>
<tr>
<td><code>memcpy</code></td>
<td><code>strcat</code></td>
<td><code>strnset</code></td>
</tr>
</tbody>
</table>

You can control the inlining of each of these functions with the `#pragma intrinsic`. For example,

```
#pragma intrinsic strcpy
```

causes the compiler to generate code for `strcpy` in your function.

```
#pragma intrinsic -strcpy
```

prevents the compiler from inlining `strcpy`. By using these pragmas in a file, you can override the command-line switches or IDE options used to compile that file.

When inlining any intrinsic function, you must include a prototype for that function before you use it. This is because, when inlining, the compiler actually creates a macro that renames the inlined function to a function that the compiler internally recognizes. In the above example, the compiler creates this macro:

```
#define strcpy __strcpy__
```

The compiler recognizes calls to functions with two leading and two trailing underscores and tries to match the prototype of that function against its own internally stored prototype. If you did not supply a prototype, or the prototype you supplied does not match the compiler’s internal prototype, the compiler rejects the attempt to inline that function and generates an error. Prototypes are provided in the standard header files (that is, `string.h`, `stdlib.h`, and so on).
The command-line compiler included in the Borland C++ product introduces a new calling convention, named \_fastcall. Functions declared using this modifier expect parameters to be passed in registers.

The compiler treats this calling convention as a new language specifier, along the lines of \_cdecl and \_pascal. Functions declared with either of these two language modifiers cannot have the \_fastcall modifier because both \_cdecl and \_pascal functions also use the stack to pass parameters. Likewise, the \_fastcall modifier cannot be used together with \_export. The compiler generates a warning if you try to mix functions of these types or if you use the \_fastcall modifier in a situation that might cause an error.

The compiler uses the rules given in Table A.2 when deciding which parameters the program is to pass in registers. A maximum of three parameters can be passed in registers to any one function. You should not assume that the assignment of registers reflects the ordering of the parameters to a function.

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>char (signed and unsigned)</td>
<td>AL, DL, BL</td>
</tr>
<tr>
<td>short (signed and unsigned)</td>
<td>AX, DX, BX</td>
</tr>
<tr>
<td>int and long (signed and unsigned)</td>
<td>EAX, EDX, EBX</td>
</tr>
<tr>
<td>pointer</td>
<td>EAX, EDX, EBX</td>
</tr>
</tbody>
</table>

Union, structure, and floating-point (float, double, and long double) parameters are pushed on the stack.

When your application calls a function using the \_fastcall calling convention, the called function automatically saves the R0, R1, and R2 floating-point registers (or the equivalent if you’re using the floating-point emulator) when called. It also restores them when the function returns. This lets the compiler allocate variables to these registers for the life of the function.

A function uses the \_fastcall calling convention when it is declared with the \_fastcall keyword or compiled with the \-pr option or Compiler | Code Generation Options | Register setting turned on.
Functions declared with the __fastcall modifier have different names than their non-__fastcall counterparts. The compiler prefixes the __fastcall function name with an @. This prefix applies to both unmangled C function names and to mangled C++ function names.
Editor reference

The tables in this appendix list all available command keystrokes. Most of these commands need no explanation. Those that do are described in the text following Table B.1.

Table B.1
Editing commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cursor movement commands</strong></td>
<td></td>
</tr>
<tr>
<td>Character left</td>
<td>←</td>
</tr>
<tr>
<td>Character right</td>
<td>→</td>
</tr>
<tr>
<td>Word left</td>
<td>Ctrl+ ←</td>
</tr>
<tr>
<td>Word right</td>
<td>Ctrl+ →</td>
</tr>
<tr>
<td>Line up</td>
<td>↑</td>
</tr>
<tr>
<td>Line down</td>
<td>↓</td>
</tr>
<tr>
<td>Scroll up one line</td>
<td>Ctrl+W</td>
</tr>
<tr>
<td>Scroll down one line</td>
<td>Ctrl+Z</td>
</tr>
<tr>
<td>Page up</td>
<td>PgUp</td>
</tr>
<tr>
<td>Page down</td>
<td>PgDn</td>
</tr>
<tr>
<td>Beginning of line</td>
<td>Home</td>
</tr>
<tr>
<td>End of line</td>
<td>Ctrl+Q S</td>
</tr>
<tr>
<td>Top of window</td>
<td>Ctrl+Q D</td>
</tr>
<tr>
<td>Bottom of window</td>
<td>Ctrl+Q E</td>
</tr>
<tr>
<td>Top of file</td>
<td>Ctrl+Q X</td>
</tr>
<tr>
<td>Bottom of file</td>
<td>Ctrl+Q C</td>
</tr>
<tr>
<td>Move to previous position</td>
<td>Ctrl+Q P</td>
</tr>
<tr>
<td>Move current line to top of window</td>
<td>Ctrl+Q T</td>
</tr>
<tr>
<td>Move current line to bottom of window</td>
<td>Ctrl+Q U</td>
</tr>
</tbody>
</table>

**Insert and delete commands**
Delete character | Del |
<table>
<thead>
<tr>
<th>Command</th>
<th>Key Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete character to left</td>
<td>Backspace, Shift+Tab</td>
</tr>
<tr>
<td>Delete word to left</td>
<td>Ctrl+Backspace</td>
</tr>
<tr>
<td>Smart tab</td>
<td>Ctrl+Tab</td>
</tr>
<tr>
<td>Delete line</td>
<td>Ctrl+Y</td>
</tr>
<tr>
<td>Delete to end of line</td>
<td>Ctrl+Q Y, Shift+Ctrl+Y</td>
</tr>
<tr>
<td>Delete word</td>
<td>Ctrl+T</td>
</tr>
<tr>
<td>Insert newline</td>
<td>Enter</td>
</tr>
<tr>
<td>Insert tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Insert line</td>
<td>Ctrl+N, Ctrl+O O</td>
</tr>
<tr>
<td>Insert mode on/off</td>
<td>Ins</td>
</tr>
</tbody>
</table>

**Block commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Key Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move to beginning of block</td>
<td>Ctrl+Q B</td>
</tr>
<tr>
<td>Move to end of block</td>
<td>Ctrl+Q K</td>
</tr>
<tr>
<td>Set inclusive block</td>
<td>Ctrl+K A</td>
</tr>
<tr>
<td>Set beginning of block</td>
<td>Ctrl+K B</td>
</tr>
<tr>
<td>Set end of block</td>
<td>Ctrl+K K</td>
</tr>
<tr>
<td>Set line block</td>
<td>Ctrl+K X</td>
</tr>
<tr>
<td>Set column block</td>
<td>Ctrl+K G</td>
</tr>
<tr>
<td>Set regular block</td>
<td>Ctrl+K M</td>
</tr>
<tr>
<td>Hide/Show block</td>
<td>Ctrl+K H</td>
</tr>
<tr>
<td>Mark line</td>
<td>Ctrl+K L</td>
</tr>
<tr>
<td>Print selected block</td>
<td>Ctrl+K P</td>
</tr>
<tr>
<td>Mark word</td>
<td>Ctrl+K T</td>
</tr>
<tr>
<td>Delete block</td>
<td>Ctrl+K Y</td>
</tr>
<tr>
<td>Copy block</td>
<td>Ctrl+K C</td>
</tr>
<tr>
<td>Move block</td>
<td>Ctrl+K V</td>
</tr>
<tr>
<td>Convert word to lowercase</td>
<td>Ctrl+K E</td>
</tr>
<tr>
<td>Convert word to uppercase</td>
<td>Ctrl+K F</td>
</tr>
<tr>
<td>Convert block to lowercase</td>
<td>Ctrl+K O</td>
</tr>
<tr>
<td>Convert block to uppercase</td>
<td>Ctrl+K N</td>
</tr>
<tr>
<td>Toggle case of block</td>
<td>Ctrl+Q O</td>
</tr>
<tr>
<td>Copy to Clipboard</td>
<td>Ctrl+Ins</td>
</tr>
<tr>
<td>Cut to Clipboard</td>
<td>Shift+Del</td>
</tr>
<tr>
<td>Delete block</td>
<td>Ctrl+Del</td>
</tr>
<tr>
<td>Indent block</td>
<td>Ctrl+K I, Shift+Ctrl+I</td>
</tr>
<tr>
<td>Paste from Clipboard</td>
<td>Shift+Ins</td>
</tr>
<tr>
<td>Read block from disk</td>
<td>Ctrl+K R</td>
</tr>
<tr>
<td>Unindent block</td>
<td>Ctrl+K U, Shift+Ctrl+U</td>
</tr>
<tr>
<td>Write block to disk</td>
<td>Ctrl+K W</td>
</tr>
</tbody>
</table>
Table B.1: Editing commands (continued)

| Extending selected blocks                      | Shift+ ←  |
| Left one character                             | Shift+ ↑  |
| Right one character                            | Shift+ ↓  |
| End of line                                    | Shift+End |
| Beginning of line                              | Shift+Home|
| Same column on next line                       | Shift+    |
| Same column on previous line                   | Shift+    |
| One page down                                  | Shift+PgDn|
| One page up                                    | Shift+PgUp|
| Left one word                                  | Shift+Ctrl+ ←|
| Right one word                                 | Shift+Ctrl+ →|
| End of file                                    | Shift+Ctrl+End|
| Beginning of file                              | Shift+Ctrl+Home|

| Other editing commands                         | Ctrl+O 1  |
| Autoindent mode on/off                         | Ctrl+O R  |
| Cursor through tabs on/off                     | Alt+F4    |
| Exit the IDE                                   | Ctrl+Q n *|
| Find place marker                              | Ctrl n *  |
| Help                                           | F1        |
| Help index                                     | Shift+F1  |
| Insert control character                       | Ctrl+P**  |
| Optimal fill mode on/off                       | Ctrl+O F  |
| Pair matching                                  | Ctrl+Q [, Ctrl+Q ],  |
| Playback keyboard macro                        | Ctrl+Shift+P|
| Redo                                           | Alt+Shift+Backspace|
| Save file                                      | Ctrl+K S  |
| Search                                         | Ctrl+Q F  |
| Search again                                   | F3        |
| Search and replace                             | Ctrl+Q A  |
| Search incrementally                           | Ctrl+S    |
| Set marker                                     | Ctrl+K n *|
| Tab mode on/off                                | Shift+Ctrl+ n *|
| Topic search help                              | Ctrl+O T  |
| Turn on syntax highlighting                    | Ctrl+F1   |
| Turn off syntax highlighting                   | Ctrl+O C  |
| Toggle keyboard macro                          | Ctrl+O N  |
| recording on and off                           | Ctrl+Shift+R|
| Undo                                           | Alt+Backspace|
| Unindent mode on/off                           | Ctrl+O U  |

*  n represents a number from 0 to 9.
** Enter control characters by first pressing Ctrl+P, then pressing the desired control character.
A block of text is any amount of text, from a single character to hundreds of lines, that is selected on your screen. There can be only one block in a window at a time. You can select a block several ways:

- Drag with your mouse while holding the left button.
- Hold down Shift while moving your cursor with the arrow keys.
- Double-click a word.
- Press Ctrl+KB at the beginning of the block and Ctrl+KK at the end of the block.

Once selected, the block can be copied, moved, deleted, or written to a file. You can use the Edit menu commands to perform these operations or you can use the keyboard commands listed in the following table.

When you choose Edit I Copy or press Ctrl+Ins, the selected block is copied to the Clipboard. When you choose Edit I Paste or Shift+Ins, the block held in the Clipboard is pasted at the current cursor position. The selected text remains unchanged and is no longer selected.

If you choose Edit I Cut or press Shift+Del, the selected block is moved from its original position to the Clipboard. It is pasted at the current cursor position when you choose the Paste command.

Table B.2: Block commands in depth

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy block</td>
<td>Ctrl+Ins</td>
<td>Copies a previously selected block to the Clipboard and, after you move your cursor to where you want the text to appear, pastes it to the new cursor position. The original block is unchanged. If no block is selected, nothing happens.</td>
</tr>
<tr>
<td></td>
<td>Shift+Ins</td>
<td></td>
</tr>
<tr>
<td>Copy text</td>
<td>Ctrl+Ins</td>
<td>Copies selected text to the Clipboard.</td>
</tr>
<tr>
<td>Cut text</td>
<td>Shift+Del</td>
<td>Cuts selected text to the Clipboard.</td>
</tr>
<tr>
<td>Delete block</td>
<td>Ctrl+Del</td>
<td>Deletes a selected block. You can &quot;undelete&quot; a block with Undo.</td>
</tr>
<tr>
<td>Move block</td>
<td>Shift+Del</td>
<td>Moves a previously selected block from its original position to the Clipboard and, after you move your cursor to where you want the text to appear, pastes it to the new cursor position. The block disappears from its original position. If no block is marked, nothing happens.</td>
</tr>
<tr>
<td></td>
<td>Shift+Ins</td>
<td></td>
</tr>
<tr>
<td>Paste from Clipboard</td>
<td>Shift+Ins</td>
<td>Pastes the contents of the Clipboard.</td>
</tr>
<tr>
<td>Read block from disk</td>
<td>Ctrl+KR</td>
<td>Reads a disk file into the current text at the cursor position exactly as if it were a block. The text read is then selected as a block. When this command is issued, you are prompted for the name of the file to read. You can use wildcards to select a file to read; a directory is displayed. The file specified can be any legal file name.</td>
</tr>
</tbody>
</table>
Table B.2: Block commands in depth (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write block to disk</td>
<td>Ctrl+K W</td>
<td>Writes a selected block to a file. When you give this command, you are prompted for the name of the file to write to. The file can be given any legal name (the default extension is CPP). If you prefer to use a file name without an extension, append a period to the end of its name.</td>
</tr>
</tbody>
</table>

If you have used Borland editors in the past, you might prefer to use the block commands listed in the following table.

Table B.3: Borland-style block commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set beginning of block</td>
<td>Ctrl+K B</td>
<td>Begins selection of text.</td>
</tr>
<tr>
<td>Set end of block</td>
<td>Ctrl+K K</td>
<td>Ends selection of text.</td>
</tr>
<tr>
<td>Hide/show selected text</td>
<td>Ctrl+K H</td>
<td>Alternately displays and hides selected text.</td>
</tr>
<tr>
<td>Inclusive block</td>
<td>Ctrl+K A</td>
<td>With inclusive blocks on, the cursor position is included as part of the block. To turn off inclusive blocks, turn on columnar, line, or regular blocks.</td>
</tr>
<tr>
<td>Turn on columnar blocks</td>
<td>Ctrl+K G</td>
<td>Turns on column blocking. To turn off columnar blocks, turn on inclusive, regular, or line blocks.</td>
</tr>
<tr>
<td>Turn on regular blocks</td>
<td>Ctrl+K M</td>
<td>Turns on regular blocking. To turn off regular blocks, turn on inclusive or columnar blocks.</td>
</tr>
<tr>
<td>Turn on line blocks</td>
<td>Ctrl+K X</td>
<td>Turns on line blocking. To turn off line blocks, turn on inclusive, columnar, or regular blocks.</td>
</tr>
<tr>
<td>Copy selected text to the cursor</td>
<td>Ctrl+K C</td>
<td>Copies the selected text to the position of the cursor. Useful only with the Persistent Block option.</td>
</tr>
<tr>
<td>Move selected text to the cursor</td>
<td>Ctrl+K V</td>
<td>Moves the selected text to the position of the cursor. Useful only with the Persistent Block option.</td>
</tr>
</tbody>
</table>

* Selected text is highlighted only if both the beginning and end have been set and the beginning comes before the end.

Other editing commands

The next table describes other editing commands in more detail. The table is arranged alphabetically by command name.

Table B.4: Other editor commands in depth

<table>
<thead>
<tr>
<th>Command</th>
<th>Keys</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autoindent</td>
<td>Ctrl+O I</td>
<td>Toggles the automatic indenting of successive lines. You can also use the Autoindent Mode setting in the Environment</td>
</tr>
<tr>
<td>Cursor through tabs</td>
<td>Ctrl+O R</td>
<td>The arrow keys move the cursor to the middle of tabs when this option is on; otherwise the cursor jumps several columns over multiple tabs. Ctrl+O R is a toggle.</td>
</tr>
<tr>
<td>Command</td>
<td>Key(s)</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Find place marker</td>
<td>Ctrl+n*</td>
<td>Finds up to 10 place markers (n can be any number in the range 0 to 9) in text. Move the cursor to any previously set marker by pressing Ctrl+Q and the marker number.</td>
</tr>
<tr>
<td></td>
<td>Ctrl+Q n*</td>
<td></td>
</tr>
<tr>
<td>Optimal fill</td>
<td>Ctrl+OF</td>
<td>Toggles optimal fill. Optimal fill begins every line with the minimum number of characters possible, using tabs and spaces as necessary. This produces lines with fewer characters.</td>
</tr>
<tr>
<td>Play back keyboard macro</td>
<td>Ctrl+Shift+P</td>
<td>Plays back previously recorded keyboard macro.</td>
</tr>
<tr>
<td>Record keyboard macro</td>
<td>Ctrl+Shift+R</td>
<td>Begin recording keyboard macro. After pressing Ctrl+Shift+R, the editor remembers all keystrokes you press. To turn off record mode and return to regular editing, press Ctrl+Shift+R again. Plays back the macro you just recorded by pressing Ctrl+Shift+P.</td>
</tr>
<tr>
<td>Search incrementally</td>
<td>Ctrl+S</td>
<td>Searches for a string as you input it. As you type, the selections in the list change to match the characters you have typed. For example, if you are searching for the word search, as you type s, the cursor goes to the next word that begins with s. When you press e, if the current word does not begin with se, the cursor moves to the next word that does, and so on.</td>
</tr>
<tr>
<td>Set place</td>
<td>Shift+Ctrl n*</td>
<td>Mark up to 10 places in text. After marking your location, you can work elsewhere in the file and then easily return to your marked location by using the Find Place Marker command (being sure to use the same marker number). You can have 10 places marked in each window.</td>
</tr>
<tr>
<td></td>
<td>Ctrl+Kn*</td>
<td></td>
</tr>
<tr>
<td>Show previous error</td>
<td>Alt+F7</td>
<td>Moves the cursor to the location of the previous error or warning message. This command is available only if there are messages in the Transcript window that have associated line numbers.</td>
</tr>
<tr>
<td>Show next error</td>
<td>Alt+F8</td>
<td>Moves the cursor to the location of the next error or warning message. This command is available only if there are messages in the Transcript window that have associated line numbers.</td>
</tr>
<tr>
<td>Tab mode</td>
<td>Ctrl+OT</td>
<td>Toggles Tab mode. You can specify the use of true tab characters in the IDE with the Use Tab Character setting in the Environment</td>
</tr>
<tr>
<td>Toggle block case</td>
<td>Ctrl+QO</td>
<td>Switches all uppercase letters in the block to lowercase and all from lowercase letters to uppercase. For example, HeLIO becomes hElLo.</td>
</tr>
<tr>
<td>Unindent</td>
<td>Ctrl+OU</td>
<td>Toggles Unindent. You can also use the Backspace Unindent setting in the Environment</td>
</tr>
</tbody>
</table>

* n represents a number from 0 to 9.
Precompiled headers

Borland C++ can generate and subsequently use precompiled headers for your projects. Precompiled headers can greatly speed up compilation times.

How they work

When compiling large C and C++ programs, the compiler can spend up to half its time parsing header files. When the compiler parses a header file, it enters declarations and definitions into its symbol table. If 10 of your source files include the same header file, this header file is parsed 10 times, producing the same symbol table every time.

Precompiled header files cut this process short. During one compilation, the compiler stores an image of the symbol table on disk in a file called BCDEF.CSM by default. (BCDEF.CSM is stored in the same directory as the compiler.) Later, when the same source file (or another source file that includes the same header files) is compiled again, the compiler reloads BCDEF.CSM from disk instead of parsing all the header files again. Directly loading the symbol table from disk is over 10 times faster than parsing the text of the header files.

Borland C++ uses precompiled headers only if the second compilation uses one or more of the same header files as the first one, and if several other things, like compiler options, defined macros, and so on, are also identical.

If, while compiling a source file, Borland C++ discovers that the first #include statements are identical to those of a previous compilation (of the same source or a different source), it loads the binary image for those #include statements, and parses the remaining statements.

Use of precompiled headers for a given module is an all or nothing deal: the precompiled header file is not updated for that module if compilation of any included header file fails.
Drawbacks

When Borland C++ uses precompiled headers, BCDEF.CSM can become very large, because it contains symbol table images for all sets of includes encountered in your sources. You can reduce the size of this file; see “Optimizing precompiled headers” on page 149.

If a header contains any code, then it can’t be precompiled. For example, although C++ class definitions can appear in header files, you should take care that only member functions that are inline are defined in the header; heed warnings such as “Functions containing for are not expanded inline”.

Using precompiled headers

You can control the use of precompiled headers in any of the following ways:

- From within the IDE, using the Compiler | Code Generation subsection of the Settings notebook (see page 63). The IDE bases the name of the precompiled header file on the project name, creating PRJ_NAME.CSM.
- From the command line using the -H, -H=filename, and -Hu options (see page 120).
- From within your code using the pragmas hdrfile and hdrstop (see Chapter 5 in the Programmer’s Guide).

Setting file names

The compiler uses just one file to store all precompiled headers. The default file name is BCDEF.CSM. You can explicitly set the name with the -H=filename command-line option or the #pragma hdrfile directive.

Caution!

You might notice that your .CSM file is smaller than it should be. If this happens, the compiler might have run out of disk space when writing to the .CSM file. When this happens, the compiler deletes the .CSM in order to make room for the .OBJ file, then starts creating a new (and therefore shorter) .CSM file. If this happens, just free up some disk space before compiling.

Establishing identity

The following conditions need to be identical for a previously generated precompiled header to be loaded for a subsequent compilation. The second or later source file must

- Have the same set of include files in the same order.
- Have the same macros defined to identical values.
- Use the same language (C or C++).
Use header files with identical time stamps; these header files can be included either directly or indirectly.

In addition, the subsequent source file must be compiled with the same settings for the following options:

- Underscores on externs (-u)
- Maximum identifier length (-in)
- Target OS/2 or PM (-W or -Wx)
- Word alignment (-a)
- Default calling convention (-p)
- Treatment of enums as ints (-b)
- Default unsigned char (-K)
- Virtual table control (-Vx)
- C++ member pointer control (-Vmx)
- Debug information (-v)
- Inline function expansion (-vi)
- Keyword control (-A)

For Borland C++ to most efficiently compile using precompiled headers, follow these rules:

- Arrange the header files in the same sequence in all source files.
- Put the largest header files first.
- Prime BCDEF.CSM with often-used initial sequences of header files.
- Use `#pragma hdrstop` to terminate the list of header files at well-chosen places. This lets you make the list of header files in different sources look similar to the compiler. Chapter 5 in the *Programmer's Guide* describes `#pragma hdrstop` in more detail.

For example, if you had two source files, ASOURCE.C and BSOURCE.C, which both included pm.h and myhdr.h:

```c
ASOURCE.C:
#include <pm.h>
#include "myhdr.h"
#include "xxx.h"
<...>

BSOURCE.C:
#include "zz.h"
#include <string.h>
#include "myhdr.h"
#include <PM.h>
<...>
```
Rearrange the beginning of BSOURCE.C to:

Revised
BSOURCE.C:

```c
#include <PM.h>
#include "myhdr.h"
#include "zz.h"
#include <string.h>
<...>
```

Note that windows.h and myhdr.h are in the same order in BSOURCE.C as they are in ASOURCE.C. You could also make a new source called PREFIX.C containing only the header files, like this:

```c
PREFIX.C

#include <PM.h>
#include "myhdr.h"
```

If you compile PREFIX.C first (or insert a `#pragma hdrstop` in both ASOURCE.C and BSOURCE.C after the `#include "myhdr.h"` statement) the net effect is that after the initial compilation of PREFIX.C, both ASOURCE.C and BSOURCE.C are able to load the symbol table produced by PREFIX.C. The compiler then needs only to parse xxx.h for ASOURCE.C and zz.h and string.h for BSOURCE.C.
Using the Browser

Browsing through your code

The PM IDE has a useful programming tool, the Browser. It lets you explore the objects in your programs and much more. Even if the applications you develop don’t use object-oriented programming, you’ll still find the Browser an extremely valuable tool. Taking full advantage of the PM graphical environment, the Browser lets you browse through object hierarchies, functions, variables, and so on. With the Browser, you can:

- Graphically view the object hierarchies in your application, then select the object of your choice and view the functions and other symbols it contains.
- List the global symbols your program uses, then select one and view its declaration, list all references to it in your program, or go to where it is declared in your source code.
- Select a symbol in your source code, then view its details at the click of the right mouse button.

Before you use the Browser, be sure to check these options in the Compiler | Code Generation subsection of the Settings notebook:

- Debug info in OBJs
- Browser info in OBJs

You need to also check these options in the Linker | Options subsection of the Settings notebook:

- Include debug info

To activate the Browser, choose Classes or Globals on the Search menu. You can also place your cursor on a symbol in your code and choose Search | Symbol At Cursor to bring up the Browser. If the program in the current window or the primary file hasn’t been compiled yet, the IDE will display an error message Error: .EXE file not found.

If your program compiles, makes, or builds successfully once, you make some changes to your code, and your next compilation fails, you can still
browse through your application as it existed at the last successful compilation.

The Browser has a SpeedBar at the top of the Browser window. Choose any SpeedBar button by clicking it with your mouse or using a hot key. By choosing a button or an associated hot key, you tell the Browser to perform some action. These are the buttons you will see, their keyboard equivalents, and the action they perform:

<table>
<thead>
<tr>
<th>Button</th>
<th>Action Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Help</td>
</tr>
<tr>
<td><code>Ctrl+Shift+G</code></td>
<td>Go to the source code for the selected item</td>
</tr>
<tr>
<td><code>Ctrl+Shift+B</code></td>
<td>Browse (view the details of) the selected item</td>
</tr>
<tr>
<td><code>Ctrl+V</code></td>
<td>View the previous browser window</td>
</tr>
<tr>
<td><code>Ctrl+C</code></td>
<td>Display an overview of the object hierarchy</td>
</tr>
<tr>
<td><code>Ctrl+Shift+O</code></td>
<td></td>
</tr>
<tr>
<td><code>Ctrl+R</code></td>
<td>List all references of a symbol</td>
</tr>
<tr>
<td><code>Ctrl+Shift+R</code></td>
<td></td>
</tr>
<tr>
<td><code>Ctrl+W</code></td>
<td>Toggles browser between single and multiple window mode</td>
</tr>
</tbody>
</table>

The last two buttons shown are actually two different views of the same button. The first time you use the Browser, you’ll see the Single Window button. Click it and it is replaced with the Multiple Window button.
When you choose the Single Window button and begin browsing, a new browser window replaces the previous window each time you perform a new browsing action. When you choose the Multiple Window button, Browser windows remain onscreen until you close them.

You can quickly reverse the action of the Window buttons; hold down Shift as you select your next browse action. For example, if the Multiple Window button is displayed, when you hold down Shift, the next browser window you open replaces the current one.

The Browser lets you see the “big picture,” the object hierarchies in your application, as well as the small details. To activate the Browser and see your objects displayed graphically, choose Search | Classes. The Browser draws your objects and shows their ancestor-descendant relationships in a horizontal tree. The red lines in the hierarchy help you see the immediate ancestor-descendant relationships of the currently selected object more clearly.

To see more detail about a particular object, double-click it. If you aren’t using a mouse, select the object by using your arrow cursor keys and press Enter. The Browser lists the symbols (the functions, variables, and so on) used in the object.
One or more letters appear to the left of each symbol in the object. The letters describe what kind of symbol it is.

You can change several filter settings at once. Drag your mouse over the cells you want to select in the Filters matrix.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Function</td>
</tr>
<tr>
<td>T</td>
<td>Type</td>
</tr>
<tr>
<td>V</td>
<td>Variable</td>
</tr>
<tr>
<td>I</td>
<td>Inherited from an ancestor</td>
</tr>
<tr>
<td>V</td>
<td>Virtual method</td>
</tr>
</tbody>
</table>

The same letters that identify the kind of symbol appear in a Filters matrix at the bottom of the Browser window. You can use filters to select the type of symbols you want to see listed.

The Filters matrix has a column for each letter; the letter can appear in the top or bottom row of this column.

To view all instances of a particular type of symbol, click the top cell of the letter's column. For example, to view all the variables in the currently selected object, click the top cell in the V column. All the variables used in the object appear.

To hide all instances of a particular type of symbol, click the bottom cell of the letter's column. For example, to view only the functions in an object, you need to hide all the variables. Click the bottom cell in the V column, and click the top cell in the F column.

In some cases more than one letter appears next to a symbol. The second letter appears just after the letter identifying the type of symbol and further describes the symbol:
• I indicates an inherited symbol
• v indicates a virtual symbol

Use one of these methods to see the declaration of a particular listed symbol:

• Double-click the symbol.
• Select the symbol and click the Browse button or press Ctrl+Shift+B.
• Select the symbol and press Enter.

If you are browsing in single-window mode (the Window button displays only one window on the SpeedBar), and you want to return to a higher level, click the Previous Browser Window button or press Ctrl+V.

Although it's very easy to use the SpeedBar to choose between single- and multiple-window mode, you can do the same thing using Ctrl+W.

Choose Search | Globals to open a window that lists every global symbol in your application in alphabetical order.

Click the symbol you want more information about or use your cursor keys to select it. A Search input box at the bottom of the window lets you quickly search through the list of global symbols by typing the first few letters of the symbol’s name. As you type, the highlight bar in the list box moves to a symbol that matches the typed characters.

Once you select the global symbol you are interested in, you can

• Choose the Browse button to see the declaration of the symbol.
• Choose the Go To Source Code button to see how the symbol is declared in the source code.
• Choose the Reference button to see a list of references to the symbol. To go to the actual reference in the code, double-click the reference in the reference list, or select it and press Enter.

You can also browse any symbol in your code without viewing object hierarchies or lists of symbols first. Just place the cursor on the symbol you wish to browse (either by clicking it or using the arrow keys to move the cursor) and choose the Search | Symbol At Cursor menu command.

If the symbol you select to browse is a structured type, the Browser shows you all the symbols in the scope of that type. You can then choose to inspect any of these further. For example, if you choose an object type, you’ll see all the symbols listed that are within the scope of the object.
Index

<> (angle brackets)
   in #include directive 82
; (semicolons) in directory path names 83
~ (tilde) in transfer program names 90
/b IDE option 17
/m IDE option 17
#pragma
   hdrstop 149

A
~a BCC option (align integers) 113
~A BCC option (ANSI keywords) 116
About Borland C++ command 57
Action On Messages settings 76
activating
   the PM Browser 151
activating, menu bar 18
active window  See windows, active
Add button 51
Add Item command 51, 94
Address And Type In Locals setting 79
aligning words and integers 63, 113
American National Standards Institute  See ANSI
angle brackets (<>)
   in #include directive 82
ANSI
   C standard 3
   compatible code 116
   floating point conversion rules 113
   keywords
      using only 68
      violations 118
   keywords (Borland C++, implementation-specific)
      option 116
ANSI violations
   settings 70
   warnings 70
Argument Names In Stack setting 78, 79, 80, 81
Argument Names setting 79
Argument Values In Stack setting 79, 80, 81
Argument Values setting 79
arguments variable list 115
Arrange Icons command 53
arrows in dialog boxes 26
.ASM files  See assembly language
assembler
   compile via 64
   default name 120
   source file setting 63
assembly language
   assembling from the command line 105
   compiling 120
   directory 125
   inline routines 120
   options
      passing 121
      removing 121
   output files 121
   projects and 99
assembly level debugger  See Turbo Debugger
Assume No Pointer Aliasing setting 67
~AT BCC option (Borland C++ keywords) 116
~AU BCC option (UNIX keywords) 117
autoindent mode 143, 145
autoindent mode setting 86
automatic dependencies 71
   checking 98
   information, disabling 115
AutoSave settings 84

B
~b BCC option (allocate whole word for enums) 113
~B BCC option (process inline assembler code) 120
Backspace Unindents setting 86
backward searching 43
.BAK files 85
bar, title 24
Base Address setting 73
BBS segment  See segments
BCDEF.CSM 121, 147, 148,  See also .CSM files
BG setting 89
BIX, JOIN BORLAND 8
block
    column 142, 145
    convert to lowercase 142
    convert to uppercase 142
    copy 142, 144
    Borland-style 145
    current line 142
    cut 144
    defined 144
    delete 142, 144
    extending 143
    hide and show 142
    Borland-style 145
    inclusive 145
    indent 142
    line 145
    move 142, 144
    Borland-style 145
    move to beginning/end of 142
    print 142
    read from disk 142, 144
    regular 145
    set beginning/end of 142
    Borland-style 145
    set column 142, 145
    set inclusive 142
    set line 145
    set regular 142, 145
    toggle case 142, 146
    unindent 142
    write to disk 142, 144
block operations (editor) See editing, block operations
blocks, text See editing, block operations
Bold setting 89
boldface text 30, 88
Borland
    contacting 8
    Borland, contacting 8-9
Borland C++
    installing 11-13
Break command 47
Break Make On
    Make dialog box 97
    Break Make On setting 70
breaking program execution 47
breakpoints See also debugging; watch expressions
    clearing 48
    saving 84
    setting 48
Breakpoints command 48
Breakpoints setting 84
Browser
    buttons on the SpeedBar 152
    hot keys 152
    in the PM IDE 151-155
    activating 151
    filters 154
    SpeedBar 152
browser
    storing information 32
Browser info in OBJs 151
Browser Info In OBJs setting 32
browsing
    in the PM IDE 151-155
    objects 153
    structured types 155
    symbols in code 155
    through global symbols 155
build
    IDE option (/b) 17
Build All command 46
bulletin board, Borland 46
buttons
    Browser 152
    Change All 43
    choosing 26
    in dialog boxes 26
    radio 27
C
C++
    exception handling 124
    External Virtual Tables
        IDE setting 66
    Local Virtual Tables
        IDE setting 66
    member functions 73
    member pointers 65
    options 65
    Public Virtual Tables
        IDE setting 66
settings
  Member Pointers 65
  Options 65
  Virtual Tables 66
  Warnings 70
Smart Virtual Tables
  IDE setting 66
  virtual tables 66
  warnings 70
-c BCC option (compile but don’t link) 120
-C BCC option (nested comments) 117
C calling conventions 65, 114
Call Stack command 48
Call Stack View Local Options settings 79
Call Stack View setting 76
Call Stack Will Show setting 79
calling convention
  __fastcall 115
  Register 115
calling conventions
  __cdecl 65, 138
  __fastcall 65, 138, 139
  __pascal 65, 138
  __stdcall 65
  __cdecl 114
  __pascal 114
  __stdcall 114
C 65, 114
Pascal 65, 114
Register 65
Standard 65, 114
Calling Conventions settings 65
Cancel button 26
SCAP EDIT macro 90
Cascade command 53
Case-Sensitive Exports setting 73
Case-Sensitive Library setting 75
Case-Sensitive Link setting 73
case sensitivity
  exports setting 73
  librarian setting 75
  linking with 73
  module definition file and 73
  searches in 41
  __cdecl calling convention 65, 138
  __cdecl
  command-line option 114
  __cdecl calling convention 114
  __cdecl statement 115
.CFG files See configuration files
Change All button 43
changing and saving settings 62
characters
  char data type See data types, char
delete 141
  tab printing 39
Check Auto-dependencies setting 71
check boxes 27
classes See also structures
  browsing 32
  container class libraries 74
  inheritance 65
  inspecting 32
Clear command 41, 144
  hot key 23
Clipboard 40, 144
  copy to 142
  cut to 142
  paste from 142, 144
  saving across sessions 85
Close All command 53
Close command
  hot key 22
Close Project command 51
closing the Settings notebook 59
code generation
  command-line compiler options 113
debugging information 32
  IDE settings 32, 47, 63
Code Generation Options settings 32, 47, 63
code page 84
Code Page setting 84
Code Sample setting 89
code segment
  group 120
  naming and renaming 119
colors
  background 30
  changing IDE text 30
  changing text 30, 88
  foreground 30
columns
  numbers 24
command-line compiler
  options
    warnings (–wxxx) 117-119
commands  See also command-line compiler, options; individual command names
  choosing 18, 22
    with the SpeedBar 19
editor
  block operations 142, 144-145
    insert and delete 141
comments, nested 69, 117
compatibility 107
compilation 112
  assembler source output 63
    breaking 47
command line  See command-line compiler
command-line compiler options 120
configuration files  See configuration files
DLLs 125
of a multiple-thread program 125
optimizations 68
rules governing 109
speeding up 63
stopping after errors and warnings 69
to .EXE file 46
to .OBJ file 46
Compile
  command 46
    menu 46
Compile Via Assembler setting 64
compiling  See compilation
  C and C++ programs 66
CompuServe, GO BORLAND 8
configuration files 28
  command-line compiler 28, 106, 110
    creating 111
    overriding 105, 111
    priority rules 111
  contents of 28
IDE 28-30
  TCCONFIG.TC 28
  TURBOC.CFG 28, 110
configuring element colors 30
constants
  hexadecimal, too large 118
  manifest  See macros
  octal, too large 118
container class libraries 74
Container Class Libraries settings 74
Contents command 55
  hot key 23
control characters
  inserting 143
conventions
  calling 65, 114
    typographic 6
  conversions
    floating point, ANSI rules 113
    pointers, suspicious 118
coprocessors  See numeric coprocessors
Copy
  command 41
    hot key 23
copy
  block (Borland-style) 145
    to Clipboard 142
copyright information 57
CPP (preprocessor)  See The online document UTIL.DOC
.CPP files  See C++
Create Backup Files setting 85
Create Extended Dictionary setting 75
.CSM files 147, 148
  default names 148
  disk space and 148
  smaller than expected 148
Ctrl+Break key 44
Current Window setting 85
Cursor Through Tabs setting 86, 143, 144, 145
customer assistance 8-9
Cut
  command
    hot key 23
Cut command 41, 142

D
  –D BCC option (macro definitions) 112
  –d BCC option (merge literal strings) 113
data, aligning 63
data segments
  group 119, 120
  naming and renaming 119, 120
data types
  char 63, 113
default
changing 63, 113
floating point See floating point
integers See integers
Datapoints command 48
Dead Storage Elimination setting 67
Debug info in OBJs 151
Debug Info In OBJs setting 32, 64
Debug menu 47
Debug Source directory, input box 33, 47, 83
Debug Source setting 33, 47, 83
Debugger Options settings 75
Debugger settings 33, 47
debugging See also integrated debugger
breakpoints See breakpoints
Browser Info In OBJs 32
Debug Info In OBJs 32, 64
Debugger Options settings 75
Debugger settings 33, 47
exceptions 76
hot keys 23
information 44
command-line compiler option 115
excluding 52
in .EXE or .OBJ files 116
including 33, 47, 64, 72
linking 33, 47, 72
storing 32, 64
inspecting a variable 34
line numbers information 32, 64
mode
hard 75
soft 75
popups on exceptions 76
reset program 45
running to cursor 45
saving breakpoints 84
setting a breakpoint 34
setting a datapoint 34
setting a messagepoint 34
setting an exceptionpoint 34
settings
Call Stack View 76
Disassembly View 76
Local Variable View 76
PM debugging mode 75
Popup On Exception 76
Source View 76
Use Evaluator 75
source directory 33, 47, 83
stack overflow 32, 65
starting a session 44
stepping
into functions 45
over functions 45
subroutines 64, 67
watch expressions See watch expressions
watching variables 34
debugging an application 32
Debugging Options settings 64
.DEF files, import libraries and 72
default assembler 120
Default BG setting 88
default buttons 26
default extension 87
Default Extension setting 87
Default FG setting 88
#define directive
command-line compiler options 112
ganging 113
Defines setting 65
Delete Item command 52, 94
Delete setting 90
deleting
blocks 142
text (redoing/undoing) 41
$DEP() macro 71
dependencies 71
desktop
saving settings in 85
system menu 21, 22
window, arranging icons in 53
Desktop setting 84
Desktop settings 85
dialog boxes See also buttons; check boxes; list boxes; radio buttons
arrows in 26
defined 26
entering text 27
Modify/New Transfer Item 90
Preferences 145
directories
.ASM and .OBJ command-line options 125
debug source 33, 47, 83
defining 82
include files 106, 125
   example 127
libraries 126
   command-line option 106, 125
   example 127
output 82
project files 29
projects 96
semicolons in paths 83
Directories settings 82
Disassembly command 48
Disassembly View Local Options settings 77
Disassembly View setting 76, 77
disk space, running out of 148
Display ASCII In File View setting 82
Display Memory As setting 78
Display Selected Item As setting 79
Display Warnings settings 69
distribution disks
   backing up 11
   defined 11
DLLs See also import libraries
   compiling 125
   import libraries and 72
   linking 72, 125
   MAKE and 72
.DSK files
   default 29
   projects and 29
Duplicate Strings Merged setting 63
dynamic link libraries See DLLs

E
-E BCC option (assembler to use) 120
-e BCC option (EXE program name) 124
Edit See also IDE, editor
   menu 39
   windows
      loading files into 97
      setting settings 85
Edit setting 90
editing
   block operations 142, 144-145
   deleting 144
   deleting text 86
   marking 86
   overwrite 86
   reading and writing 144
   selecting blocks 39, 89
   copy and paste
      hot key 23
   cut and paste 40, 41
   hot keys 23
   pair matching See pair matching
   redoing undone text edits 41
   selecting text 39, 144
   undelete 41
   undoing text edits 41
   windows
      cursor, moving 141
   editor See IDE, editor
   Editor Files
      setting, Auto Save 84
   Editor Files setting 84
   Editor Key Bindings setting 83
   Editor Options settings 85
   Editor settings 85
   Element setting 88
   ellipsis ( ... ) 18, 26
   enumerations See enum (keyword)
   enumerations (enum)
      assigning integers to 118
      treating as integers 63, 113
Environment
   setting, Auto Save 84
   environment See IDE
   Environment setting 84
   Error Messages command 56
   errors See also warnings
      ANSI 118
      frequent 118
      messages 5
         compile time 96, 97
         removing 98
         saving 98
         searching 50
      reporting command-line compiler options 117
      show next/previous 146
      stopping on n 69
      syntax, project files 96, 97
      tracking, project files 96, 97
Errors, Stop After setting 69
Esc shortcut 26
Essentials command 56
evaluation order
   command-line compiler options 111
      in response files 110
Evaluator command 49
Evaluator Show setting 80
Evaluator View Local Options settings 80
examples
   library and include directories 127
Exception Handling compiler options 124
Exceptionpoints command 48
.EXE files
   creating 23, 46
directories 82
   linking 46
   naming 46
   user-selected name for 124
executable files See .EXE files
Exit command 39
exiting
   IDE 143
   exiting Borland C++ 22
   exiting the IDE 17
explicit library files 125
__export (keyword) 138
exported member functions 73
exports, case sensitive 73
extended dictionary setting
   librarian 75
extension keywords, ANSI and 116
Extension setting 89
External Virtual Tables
   command-line option 122

F
far virtual table segment
   naming and renaming 120
__fastcall calling convention 65, 138, 139
fastcall
   command-line option 115
__fastcall calling convention 115
Fastest Code setting 68
-ff BCC option (fast floating point) 113
FG setting 88
File Alignment setting 73
File And Numeric View Local Options settings 82
file lists
   wildcards and 38
File menu 37
File View Will Display As setting 82
FILELIST.DOC 11
files See also individual file-name extensions
   browser information in OBJs 32
   C++ See C++
   closed, reopening 53
   compiling 121
   configuration 28
   debugging information in OBJs 32, 64
dependencies in OBJs 71
desktop (.DSK)
   default 29
   projects and 29
directories
   .EXE 82
   .MAP 82
   .OBJ 82
   source 33, 47, 83
directory
   see .EXE files
ingoing
   OBJs 32
include
See include files
information in dependency checks 98
library (.LIB) See libraries
loading into editor 97
make See MAKE (program manager)
map 73, See map files
modifying 14
module definition 125
module definition files
   IMPORTS section, case-sensitive 73
new 37
NONAME 37
opening 37
out of date, recompiled 98
printing 39
project 28
README 13
response See response files
saving 38, 143
   all 38
automatically 84
with new name or path 38
source, .ASM, command-line compiler and 105
filling lines with tabs and spaces 86
filters
    transfer  See transfer filters
filters, PM Browser 154
Find command 41, See also searching
Flags setting 77, 81
floating point
   ANSI conversion rules 113
   fast 113
Follow PC setting 77
Fonts settings 87
Format Of Selected Item setting 80
Frame Registers setting 79, 81
full link map 125
Function Entries settings 78
functions  See also member functions
   browsing through PM 155
calling conventions 65, 114, 115
defined in source, going to 155
exported 73, 138
help 56
inline, precompiled headers and C++ 148
naming 139
   view details of 155
   void, returning a value 118

G

ganging
   command-line compiler options
      #define 113
      macro definition 113
defined 113, 126
IDE 126
   library and include files 126
General settings 70
Generate Assembler Source setting 63
Generate Import Library settings 72
Generate List File setting 75
Generate Makefile
   command 52
Generate Underbars setting 64
GENie, BORLAND 9
Global Optimizations setting 67
Global Register Allocation setting 67
global variables, word-aligning 113
Global Variables settings 78
-gn BCC option (stop on n warnings) 117
Go to Cursor command
   hot key 23
Go to Line Number command 44
GREP (file searcher)  See The online document
   UTIL.DOC
   wildcards in the IDE 42
Group Undo setting 86
   Undo and Redo commands and 41

H

-H BCC option (precompiled headers) 120
hardware requirements to run Borland C++ 2
hdrfile pragma 148
hdrstop pragma 148, 149, 150
header files  See also include files
   help 56
   precompiled  See precompiled headers
   searching for 126
Heap command 49
Help
   hot keys 22
   help 143
      accessing 22, 54
      button 26
      C and C++ 56
      HELPME!.DOC file 13
      hot keys 22, 23
      index 55, 143
      language 56
      links 54
      menu 54
      status line 26
      table of contents 55
      topic search 143
      using 56
Using Help command 56
windows
   closing 54
   links in 54
   opening 54
   selecting text in 55
hexadecimal numbers  See numbers, hexadecimal
Hide Windows command 49
hierarchies  See classes
hierarchy
  viewing an object 152
history lists 27
  closing 53
  saving across sessions 85
Horizontal setting 81
hot keys 37
  debugging 23
  editing 23
  help 22, 23
  menus 21, 22
  using 21

-i BCC option (identifier length) 117
-I BCC option (include files directory) 106, 125
icons, arranging 53
IDE 15
  command-line arguments in the 45
  commands
    cursor movement 141
    insert and delete 141
  customizing 14
  editor
    cursor movement 141
    fonts 87
    miscellaneous commands 145-146
    options 85
    setting defaults 85
    tabs in 86
  ganging multiple directories 126
  options 15
  starting up 15
  syntax highlighting 30, 88
BC and BCC See Borland C++; command-line
  compiler; IDE
Identifier Length 69
Identifier Length settings 69
identifiers
  Borland C++ keywords as 68, 116
    length 69
  Pascal 115
    significant length of 113, 117
    undefining 112
    underscore for 115
  image base address 73
  Image Is Based setting 73

IMPLIB (import librarian) See import libraries
$IMPLIB See import libraries
$IMPLIB macro 72
implicit library files 125
import libraries See also DLLs
  DLLs and 72
    generating 72
  include debug info 33, 47, 72
  Include Debug info in OBJs 151
#include directive See also include files
  angle brackets (<> and 126
  directories 82
  quotes and 126
Include Directories
  input box 82
include files See also header files
  command-line compiler options 126
    directories 106, 125
      multiple 127
    help 56
    projects 94, 95
    searching for 126
    user-specified 106, 125
Include Files command 52, 94
Include setting 82
Include Views setting 77
incremental search 27
indent
  automatic 86
  block 142
Index command
  Help menu 55
  hot key 23
information
  technical support 8
initialization See specific type of initialization
inline code See specific type of initialization
input boxes 27
insert
  control characters 143
  lines 142
  mode 142
  newline 142
  tab 142
Inspector command 34, 49
Inspector View Local Options settings 80
installation 11-13
integers 113, See also floating point; numbers
aligned on word boundary 113
assigning to enumeration 118
integrated development environment See IDE
debugging See debugging; integrated debugger
menus See menus
integrated environment
makes 98
intrinsic pragma 137
Italic setting 89
italicize text 30, 88

J
-`Jg` BCC options (template generation options) 123
-`Jg` options (template generation options) 66
-`jn` BCC option (stop on `n` errors) 117

K
-`k` BCC option (standard stack frame) 114
-`K` BCC option (unsigned characters) 113
K&R See Kernighan and Ritchie
Keep Messages command
toggle 98
Kernighan And Ritchie
keywords 68
Kernighan and Ritchie
keywords 117
keyboard
choosing commands with 18, 26
selecting text with 39
Keyboard command 56
keyboard macros
playing back 143, 146
recording 143, 146
keywords 68
ANSI command 116
Borland C++ 68, 116
Kernighan and Ritchie, using 117
Keywords settings 68
settings 68
UNIX, using 117

L
-L BCC option (linker options) 124
-L BCC option (object code and library directory)
106, 125
language help 56
Language Reference command 56
.LIB files See libraries
librarian
case sensitive setting 75
extended dictionary setting 75
list file setting 75
purge comments setting 75
libraries
command-line compiler options 126
counter class 74
directories 82, 125
command-line option 106, 125
multiple 127
dynamic link See DLLs
explicit and implicit 125
files 82, 106, 125
import See import libraries
linking 46
multi-thread 72
multiple thread (C2MT.LIB) 125
overriding in projects 101
rebuilding 115
searching for 126
single thread 72
standard run-time 74
user-specified 125
Library Directories input box 82
library page size 75
Library Page Size setting 75
Library setting 82
line numbers See lines, numbering
Line Numbers Debug setting 32, 64
lines
deleting 142
filling with tabs and spaces 86
inserting 142
marking 142
moving cursor to 44
numbering 24
in object files 115
information for debugging 32, 64
restoring (in editor) 41
Link command 46
Link Libraries settings 74
link map, full 125
Link Settings settings 47, 72
Link Warnings settings 74
linking
  breaking 47
  case sensitive 73
command-line compiler options 124
DLLs 72, 125
  link map, creating 125
module definition files 125
multiple-thread libraries 125
options 72
  options, from command-line compiler 124
links
  help 56
  Help windows 54
list all line references 152
list boxes 27
  file names 38
list file setting, librarian 75
Local Common Expressions setting 67
local menus
  using 19, 34
Local Options
  command 94
Local Options command 52
Local Variables View setting 76
Local Variables setting 79
Local Variables settings 78
Local Virtual Tables
  command-line option 121

M
-M BCC option (link map) 125
macros
  $CAP EDIT 90
  $DEP() 71
  $IMPLIB 72
  command-line compiler 112
  ganging 113
  _MT_ 125
  transfer 90, See transfer macros
  Turbo editor 83, See also The online document
  UTIL.DOC
MAKE (program manager)
  DLLs and 72
  IDE option (/m) 17
  integrated environment makes and 98
  stopping makes 97
Make command 46
  hot key 23
manifest constants See macros
map file 73
Map File settings 73
map files 125
  directory 82
marker
  find 143, 146
  set 143, 146
math coprocessors See numeric coprocessors
Maximize box 24
member functions
  exported 73
  inline 148
member pointers, controlling 122
Memory command 49
Memory Displays As setting 81
Memory setting 77
Memory View Follows Stack setting 81
Memory View Local Options settings 80
Memory Will Show setting 80
menu bar See also menus
menu commands
  choosing
    with the SpeedBar 19
  choosing with the keyboard 18
  choosing with the mouse 18
menus See also individual menu names
  hot keys 21, 22
  IDE 18, 22
  local 19, 34
  reference 37
  Tools 89
  with an ellipsis (...) 26
Menus command 56
Message Tracking toggle 97
Messagepoints command 48
messages
  appending 85
  removing 50
Minimal Opts setting 68
Modify/New Transfer Item dialog box 90
module definition files 125
  EXPORTS section, case-sensitive 73
monitors See also screens
mouse
  buttons
    right and left 18
  choosing commands with 18, 26
selecting text with 40
moving text See editing
  _MT_ _ macro 125
multi-thread
  Multi-thread setting 72
  programs 72
multiple listings
  command-line compiler options
    #define 113
    include and library 126
    macro definition 113
  Multiple Window button 152

N
  -n BCC option (.OBJ and .ASM directory) 125
  -N BCC option (stack overflow logic) 114
Name setting 87
names See identifiers
Names settings 70
nested comments 69, 117
Nested Comments settings 69
New command 37
New Window setting 85
Next command
  hot key 22
next error, show 146
Next Error command 50
NONAME file name 37
notebook
  Settings 51
    undoing changes 62
numbers See also floating point; integers
  hexadecimal
    constants, too large 118
  octal constants
    too large 118
  real See floating point
numeric coprocessors
  generating code for 113
Numeric Processor command 49
Numeric View Display As setting 82

O
  -o BCC option (object files) 121
.OBJ files
    browser information 32
    compiling 121
    creating 46
    debugging information 32, 64
    dependencies 71
directories 82, 125
  line numbers in 115
object
  hierarchy
    viewing an 152, 153
    view details of 153
objects
  browsing
    in the PM IDE 153
OBJs
  Browser info 151
  Debug info 151
  link info 151
OBJXREF See The online document UTIL.DOC
OK button 26
online Help See help
Open command 37
Open Project command 51
opening a file 37
opening the Settings notebook 51, 59
Optimal Fill setting 86, 143, 146
optimizations 67, 129
  command-line compiler options 116
  for speed or size 68
  Optimization settings 67
  PM applications and 68
  precompiled headers 149
  registers, usage 130
  settings 67
Optimize For settings 68
options See also specific entries (such as
  command-line compiler, options)
C++ template generation
  command-line option 123
IDE 15
linking 72
Options settings
  code generation 63
  librarian 74
linker 72

OS/2
  API documentation 2
  Clipboard 40
  commands 11
  Help system 55
  path 90
  Settings notebook 15
  version 2
  wildcards 38, 89

Out-of-line Inline Functions setting 33, 66
Output
  Directory, input box 82
  Output setting 82
  Overwrite Blocks setting 86

P
-p- BCC option (_stdcall conventions) 114
-P BCC option (C++ and C compilation) 121
-\p BCC option (Pascal calling conventions) 114
-\pr BCC option (_fastcall calling convention) 115
pair matching 143
parameter types, register usage and 138
Pascal
  identifiers 115
  _pascal calling convention 65, 138
  _pascal
    command-line option 114
  _pascal calling convention 114
Pascal calling conventions 65, 114
Paste command 41
  hot key 23
paste from Clipboard 142, 144
pasting  See editing
path names in Directories dialog box 83
-\pc BCC option (C conventions) 114
Persistent Blocks setting 86
place marker
  find 143, 146
  set 143, 146
playing back keyboard macros 143, 146
PM Debugging Mode setting 75
pointers
  suspicious conversion 118
Popup On Exception settings 76
portability
  Portability settings 70

warnings 70, 118
#pragma
  hdrfile 148
  hdrstop 148
  intrinsic 137
  warn 117
#pragma hdrstop 150
precedence
  command-line compiler options 106, 111
  response files and 110
precompiled headers 147-150
  command-line options 120
  controlling 148
  drawbacks 148
  inline member functions and 148
  optimizing use of 149
  Precompiled Headers setting 63
  rules for 148
  using, IDE 63
Preferences dialog box 145
Preferences settings 83
previous browser window 152
previous error, show 146
Previous Error command 50
Print command 39
.PRJ files  See projects
procedures  See functions
Program Target settings 71
program titles 89
Program Titles setting 89
Programmer's Platform  See IDE
programs
  ending 44
  multi-source  See projects
  multi-thread 72
  rebuilding 44, 46
  running 44
    arguments for 45
  single thread 72
  transfer, list 99
Project
  menu 51
project files 28
  contents of 28
Project Manager 44
  closing projects 51
  Include files and 52
purge comments setting
librarian 75

Q
Quit
command (IDE) 17

R
-r BCC option (register variables) 130
radio buttons 27
read block 142
README 13
rebuilding libraries 115
recording keyboard macros 143, 146
redo 143
Redo command 41
Group Undo and 41, 86
hot key 23
Register calling conventions 65, 115
Register Contents Display As setting 81
Register Layout setting 81
Register View Local Options settings 81
Register View Will Show setting 81
registers
pseudovariables, using as identifiers 116
usage and parameter types 138
variables
suppressed 130, 131
toggle 130, 131
Registers command 49
Registers setting 77, 81
registration (product)
by phone 8
Remove Messages command 50, 98
Replace command 43
replacing a file 37
requirements to run Borland C++
hardware 2
software 2
Reset command 45
resetting program 45
resize corner 24
response files
defined 110
option precedence 110
Run
  command 44
  hot key 23
  menu 44
Run Arguments command 45
Run To Cursor command 45

S
-S BCC option (produce .ASM but don’t assemble) 121
Save All command 38
Save As command 38
Save command (File menu) 38
Save command (Project command) 53
save file 143
Save Old Messages settings 85
Save settings 85
saving breakpoints 84
scope See variables
scroll bar 24
scroll bars 25
-sd BCC option (compiling DLLs) 125
-sD BCC option (module definition file name) 125
Search Again command 44
  hot key 23
Search menu 41
searching
  direction 43
  error and warning messages 50
  for include files 126
  for libraries 126
  for text 143
  in list boxes 55
  incrementally 143, 146
  origin 43
  regular expressions 42
  repeating 44
  replace and 43
  scope of 43
  search and replace 43
Segment Names settings 70
segment-naming control
  command-line compiler options 119
segments
  and pragma codeseg 119
  BSS 70
  code 70
  controlling 119
  data 70
  far data 70
  naming 70
  selecting a font 87
  selecting text 144
  semicolons (;) in directory path names 83
settings
  Action On Messages 76
  Address And Type In Locals 79
  ANSI Violations 70
  Argument Names 79
  Argument Names In Stack 78, 79, 80, 81
  Argument Values 79
  Argument Values In Stack 79, 80, 81
  Assume No Pointer Aliasing 67
  autoindent mode 86
  AutoSave 84
  Backspace Unindents 86
  Base Address 73
  BG 89
  Bold 89
  Break Make On 70
  Breakpoints 84
  Browser Info In OBJs 32
  C++ Member Pointers 65
  C++ Options 65
  C++ Virtual Tables 66
  C++ Warnings 70
  Call Stack View 76
  Call Stack View Local Options 79
  Call Stack Will Show 79
  Calling Conventions
    C 65
    Pascal 65
    Register 65
    Standard 65
  Case-Sensitive Exports 73
  Case-Sensitive Library 75
  Case-Sensitive Link 73
  changing 62
  Check Auto-dependencies 71
  Code Generation Options 32, 47, 63
  Code Page 84
  Code Sample 89
  Compile Via Assembler 64
  Container Class Libraries 74

Index
Create Backup Files 85
Create Extended Dictionary 75
Current Window 85
Cursor Through Tabs 86
Dead Storage Elimination 67
Debug Info In OBJs 32, 64
Debug Source 33, 47, 83
Debugger 33, 47
Debugger Options 75
Debugging Options 64
Default BG 88
Default Extension 87
Default FG 88
Defines 65
Delete 90
Desktop 84, 85
Directories 82
Disassembly View 76, 77
Disassembly View Local Options 77
Display ASCII In File View 82
Display Memory As 78
Display Selected Item As 79
Display Warnings 69
Duplicate Strings Merged 63
Edit 90
Editor 85
Editor Files 84
Editor Key Bindings 83
Editor Options 85
Element 88
Environment 84
Errors, Stop After 69
Evaluator Show 80
Evaluator View Local Options 80
Extension 89
Fastest Code 68
FG 88
File Alignment 73
File And Numeric View Local Options 82
File View Will Display As 82
Flags 77, 81
Follow PC 77
Fonts 87
Format Of Selected Item 80
Frame Registers 79, 81
Function Entries 78
General 70
Generate Assembler Source 63
Generate Import Library 72
Generate List File 75
Generate Underbars 64
Global Optimizations 67
Global Register Allocation 67
Global Variables 78
Group Undo 86
Horizontal 81
Identifier Length 69
Image Is Based 73
Include 82
Include Views 77
Inspector View Local Options 80
Italic 89
Keywords 68
Library 82
Library Page Size 75
Line Numbers Debug 32, 64
Link Libraries 74
Link Settings 47, 72
Link Warnings 74
Local Common Expressions 67
Local Variable View 76
Local Variables 78, 79
Map File 73
Memory 77
Memory Displays As 81
Memory View Follows Stack 81
Memory View Local Options 80
Memory Will Show 80
Minimal Opts 68
Name 87
Names 70
Nested Comments 69
New Window 85
Numeric View Display As 82
Optimal Fill 86, 143, 146
Optimization 67
Optimizations 67
Optimize For 68
Options 63, 72
Options (librarian) 74
Out-of-line Inline Functions 33, 66
Output 82
Overwrite Blocks 86
Persistent Blocks 86
PM Debugging Mode 75
Popup On Exception 76
Portability 70
Precompiled Headers 63
Preferences 83
Program Target 71
Program Titles 89
Project 84
Project Name 90
Purge Comment Records 75
Register Contents Display As 81
Register Layout 81
Register View Local Options 81
Register View Will Show 81
Registers 77, 81
Save 85
Save Old Messages 85
saving 62
Segment Names 70
Show Address And Type 79
Show Source 77
Show Symbolic 77
Show Type Information 80
Size 87
Smallest Code 68
Source File 84
Source Tracking 85
Source View 76
SpeedBar 33, 85
Stack 77, 78, 79, 80, 81
Standard Run-time Libraries 74
Standard Stack Frame 64, 67
Style 88
Syntax Highlighting 30, 86
Syntax Hilite 88
Tab Size 87
Template Generation 66
Test Stack Overflow 32, 65
Thread Options 72
Translator 90
Treat Enums As Ints 63
Type Information 79, 80
Underline 89
Unsigned Characters 63
Use C++ Compiler 66
Use Evaluator 75
Use Tab Character 86
Variable Information 78
Variables View Local Options 78
Variables View Will Display 78
Vertical 81
Warnings, Stop After 69
Watch View Local Options 79
Watch Will Show 79
Word Alignment 63, 113

Settings notebook 51
  closing 59
  Compiler section 62
  Debugger Options section 75
  Debugger section 33, 47
  Directories section 82
  Environment section 83
  Librarian section 74
  Linker section 72
  Make section 70
  opening 59
  pages 60
  sections 60
  subsections 60
  Target section 71
  Transfer section 89
  undoing changes 62
  using 59
shortcuts See hot keys
  Show Address And Type setting 79
  Show Source setting 77
  Show Symbolic setting 77
  Show Type Information setting 80
  Show Windows command 49
single thread
g  programs 72
  Single Thread setting 72
Single Window button 152
Size setting 87
  -sm BCC option (link with multiple-thread libraries) 125
Smallest Code setting 68
Smart Virtual Tables
c  command-line option 121
software requirements to run Borland C++ 2
source code, go to 155
Source command 48
source debugging settings 44
Source File setting 84
source files
  .ASM, command-line compiler and 105
directories 33, 47, 83
source-level debugger  See Turbo Debugger
Source Options settings 68
source tracking 85
Source Tracking settings 85, 97
Source View setting 76
spaces vs. tabs 86
speed, optimization 116
SpeedBar 19, 33, 85
  Browser 152
    configuring the 19
    settings 33, 85
stack
  overflow 32, 65, 114
  standard frame, generating 114
Stack setting 77, 78, 79, 80, 81
standalone librarian
  case sensitive 75
  extended dictionary 75
  list file 75
  purge comments 75
Standard calling conventions 65, 114
standard library files  See libraries
standard run-time libraries 74
Standard Run-time Libraries settings 74
Standard Stack Frame
  generating 114
  setting 64, 67
start-up and exit
  IDE 15
status line 26
__stdcall calling convention 65
  command-line option 114
__declspec calling convention 114
Step Over command 45
  hot key 23
stepping
  into functions 45
  over functions 45
strings
  duplicate, merging 63
  literal, merging 113
structures
  ANSI violations 118
  undefined 118
  zero length 118
Style setting 88
support, technical 8-9
switches  See command-line compiler, options; IDE
symbolic constants  See macros
symbolic debugger  See Turbo Debugger
symbols
  browsing in source code 155
  viewing declarations of 155
syntax
  errors, project files 96, 97
  IDE options 15
syntax highlighting 30, 86
  configuring element colors 30
    IDE 30
  setting 30, 86
  settings 88
  turning on and off 143
Syntax Hilite settings 88
system menu button 16, 17, 24
system requirements 2

T
--T- BCC option (remove assembler options) 121
Tab mode 143, 146
Tab Size setting 87
tabs
  characters, printing 39
  size of 87
  spaces vs. 86
  using in the editor 86
Tasks command 56
TCCONFIG.TC  See configuration files, IDE
TCDEF.DPR files 29
TCDEF.DSK files 29
Technical Support
  contacting 8
  technical support 8-9
TEML  See The online document UTIL.DOC
Template Generation settings 66
  External 67
  Global 67
  Smart 66
templates, generation 123
terminate and stay resident  See TSR programs
Test Stack Overflow setting 32, 65
Turbo Assembler
Turbo Debugger, described 115
Turbo Editor Macro Compiler 83, See also The online document UTIL.DOC
Turbo Editor Manual Language 83, See also The online document UTIL.DOC
Type Information setting 79, 80
typefaces used in these books 6
types
  browsing structured 155
typographic conventions 6

U
-U BCC option (undefine) 112
-u BCC option (underscores) 115
Underline setting 89
underline text 30, 88
underscores 115
  generating automatically 64, 115
undo 143
Undo command 41
  Group Undo and 41, 86
  hot key 23
unindent
  block 142
  mode 143, 146
UNIX
  keywords 68
  using 117
  porting Borland C++ files to 117
Unsigned Characters setting 63
Use C++ Compiler settings 66
Use Evaluator setting 75
Use Tab Character setting 86
user-specified library files 125
utilities See also The online document UTIL.DOC

V
-v and -Vn BCC options (C++ virtual tables) 121
-v BCC option (debugging information) 115
variable argument list 115
Variable command 48
Variable Information setting 78
variables
  automatic word-aligning 113
  register 130, 131
Variables View Local Options settings 78
Variables View Will Display setting 78
version number 57
Vertical setting 81
--vi BCC option (C++ inline functions) 116
video code page 84
View Project command 51
View Settings command 51, 59
View Transcript command 50
viewing
  declarations of symbols 155
  details of an object 153
  details of functions 155
  object hierarchy 153
  the Project window 51
  the Transcript window 50
virtual tables 66
  command-line option 121, 122
  controlling 121
  -Vm BCC options (C++ member pointers) 122

W
--wxxx BCC options (warnings) 117
warn pragma 117
warnings See also errors
  ANSI violations 70
  C++ 70, 118
  command-line options 117-119
  enabling and disabling 117
  frequent errors 118
  messages 5
  options 117-119
  portability 70, 118
  settings 69
Warnings, Stop After setting 69
Watch command 34, 49
Watch View Local Options settings 79
Watch Will Show setting 79
whole-word searching 41
wildcards 42
  GREP 42
  OS/2 38
Window menu 53

window sizing buttons
  Maximize 24
  Minimize 24
  Restore 24
windows
  active 22, 23, 24, 25, 53, 85
  cascading 53
  closed 53
    listing 53
  closing 53
Edit See Edit, windows
Help See Help, windows
menu 53
  open 53
    listing 53
Project 51
  saving across sessions 85
  source tracking 85
  system menu 22
  tiling 53
Transcript 50
  using IDE 23, 24, 25
word
  convert to lowercase 142
  convert to uppercase 142
  delete 142
  mark 142
Word Alignment setting 63, 113
write block 142
--wxxx BCC options (warning me
  117-120
  --wxxx BCC options (warnings) 1

X
--X BCC option (disable autodepe.
  -X BCC option (handle exceptions.
  information) 115

Y
--y BCC option (line numbers) 115

Z
--zV BCC options (far virtual table seg
  --zV BCC options (far virtual table seg
  --zX BCC options (code and data segm
  119-120
Borland

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Canada, Chile, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # BCP1415WW21770 • BOR 7000